JAVA Mag inkl. CD

Android User Interfaces: Grundlagen und Tipps »64

S&S

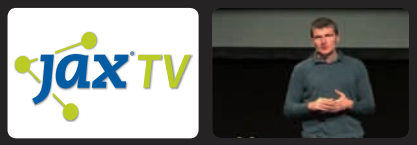
Deutschland €8,50 Österreich €9,80 Schweiz sFr 16,80 2.2011

Javamagazin™

Java • Architekturen • Web • Agile

www.javamagazin.de

CD-INHALT



ADAM BIEN
60 Minuten mit Java EE

Video von der JAX 2010 in voller Länge

HIGHLIGHTS

Java ist auch eine Insel als OpenBook

Android SDK

Clojure 1.2

WEITERE INHALTE

EasyMock 3.0

jCouchDB

ScalaTest

ScalaCheck u.v.

Alle CD-Infos ab Seite 3

Clean Code
Design Pattern Teil 3 »45

RHQ
Auf die Finger geschaut »82

jax 2011
Alle Infos im Heft »35

Clojure

Multithreading auf der JVM

Clojure: Wozu wieder eine neue Programmiersprache? »20-34

Interview mit Clojure-Gründer Rich Hickey »16

CouchDB

Bequemer Zugriff aus Java »75

TDD mit Scala

Mit guten Vorsätzen ins neue Jahr »97

Scrum But(t)

Wie viel Anpassung ist gut für Scrum? »110

Datenträger enthält Info- und Lehrprogramme gemäß §14 JuSchG

Was Sie für den Start in den Lisp-Dialekt alles wissen sollten

Clojure unter der Lupe

Clojure ist eine Sprache aus der Lisp-Familie. Als Mitglied dieser Familie erbt Clojure einige Eigenschaften, die es von Sprachen wie Java oder C unterscheiden. Der vordergründig größte Unterschied ist die Syntax. Lisp-Programme sehen für Programmierer mit C-/Java-Hintergrund einfach ungewohnt aus. Die Syntax ist aber logisch und kommt mit erheblich weniger Regeln als andere Sprachen (vielleicht ausgenommen Forth) aus. Ein weiterer Unterschied zu vielen „Mainstream"-Sprachen ist die Tatsache, dass Clojure eine funktionale Sprache ist. Funktionen sind das zentrale Element der Sprache, Seiteneffekte werden soweit möglich vermieden.

von Burkhard Neppert

Der typische Entwicklungsprozess in Lisp/Clojure vollzieht sich in einer interaktiven Umgebung, der Read-Eval-Print-Loop (REPL). Die Clojure-REPL ist ein Java-Programm, das in einer Java-VM ausgeführt wird. Sie starten eine Clojure-REPL in einer Kommandozeile durch *java -jar clojure.jar clojure.main*. Für erste Schritte ist die REPL ausreichend, komfortableres Arbeiten wird aber mit Plug-ins für Eclipse [1], NetBeans, IntelliJ sowie auf Emacs [2] möglich.

Grundlegende Dinge

Jedes Clojure-Programm besteht aus einer Reihe von Ausdrücken. Diese werden von der Read-Eval-Print-Loop (REPL) gelesen und ausgewertet und das Ergebnis der Auswertung als Resultat ausgegeben. Während der Auswertung können Seiteneffekte wie Ein-/Ausgabeoperationen auftreten. Es sind verschiedene Typen von Ausdrücken vorhanden, die nach unterschiedlichen Regeln ausgewertet werden.

Konstanten

Zahlen wie *1, 2.71828* oder Zeichenketten wie *Foo* sind konstante Ausdrücke in Clojure und werden zu sich selbst ausgewertet:

```
>> 10          ;; => 10  Kommentare in Clojure: Zeile ab „;" wird ignoriert
>> "Ideenlos" ;; => "Ideenlos"
```

Neben Zahlen und Zeichenketten existieren noch die booleschen Konstanten *true* und *false* sowie der besondere Wert *nil*, der in Java *null* entspricht. Weiter gibt es in Clojure so genannte „Keywords". Das sind Konstanten, die zu sich selbst ausgewertet werden. Die konkrete Syntax ist ein „*:*", gefolgt von einem Bezeichner, z. B. *:ein-keyword*. Eine häufige Nutzung von Keywords ist als Schlüssel für assoziative Datenstrukturen und für Multimethoden.

Symbole

Symbole sind in Clojure Bezeichner, die für andere Werte stehen. Ein Symbol wird durch *(def <symbol> <ausdruck>)* an einen Wert gebunden. Bei der Auswertung liefert das Symbol dann den Wert, an den es gebunden wurde. Die Auswertung eines Symbols, das noch nicht gebunden wurde, ist ein Fehler:

```
>> ein-symbol        ;; => Unable to resolve symbol: ein-symbol in this
                                                                  context
>> (def ein-symbol 42) ;; => #'user/ein-symbol
>> ein-symbol        ;; => 42
```

Soll das Symbol selbst als Wert verwendet werden, muss ein "'" vorangestellt werden (der Ausdruck wird „gequoted"):

```
>> frei ;; => Unable to resolve symbol: frei in this context
>> 'frei ;; => frei
>> (def unfrei 'frei)
>> unfrei => frei
```

Die durch *def* hergestellte Symbol-/Wert-Bindung ist global, d. h. überall sichtbar, sofern nicht durch andere Bindungskonstrukte das Symbol lokal an einen anderen Wert gebunden wurde (mehr dazu bei *let* und Funktionen).

Listen

Listen haben in Clojure eine doppelte Funktion: Zum einen dienen sie als einfache Datenstruktur, die beliebige Clojure-Ausdrücke (also auch wieder Listen (die wiederum Listen (mit Listen ...))) enthalten kann. Zum anderen

werden Listen bei der Auswertung als Funktionsaufrufe behandelt. Die Syntax für eine Liste ist *(<ausdruck>*)*.

Listen als Datenstruktur

Soll eine Liste als Datenstruktur verwendet werden, so ist ein „'" vor die äußerste öffnende Klammer zu setzen (im Lisp-Jargon: der Ausdruck wird „gequoted"). Dadurch wird die Liste aller nicht ausgewerteten Ausdrücke in der Liste zurückgegeben.

Ist eine Liste gewünscht, die die Ergebnisse der Auswertung der Elemente enthält, kann die Funktion *list* verwendet werden:

```
>> (def a1 "erster")
>> (def a2 "zweiter")
>> (def a3 "dritter")
>> '(a1  (a2  a2)) ;; => (a1 (a2 a3))
>> (list a1 a2 a3) ;;  => ("erster" "zweiter" "dritter")
```

Für den Umgang mit Listen enthält Clojure eine große Menge von Funktionen, hier nur die allerwichtigsten:

- *count*: Für jede Liste (und Clojure Collection) liefert *(count <liste>)* die Zahl der Elemente.
- *First*: Der Ausdruck *(first <liste>)* liefert das erste Element einer Liste und bei einer leeren Liste *'()* den besonderen Wert für „nichts", *nil*.
- *rest* und *next*: Der Ausdruck *(rest <liste>)* liefert eine Liste ohne das erste Element und bei einer leeren Liste *'()* die leere Liste. *next* verhält sich ähnlich wie *rest*, nur wird hier in Fällen, in denen *rest* die leere Liste liefert, *nil* zurückgegeben.
- *conj:* Listen werden durch *(conj <liste> <element>+)* konstruiert, wobei die Elemente an den Anfang der Liste gestellt werden:

```
>> (count '(1 2 3)) ;; => 3
>> (first '(1 2 3)) ;; => 1
>> (rest '(1 2 3)) ;; => (2 3)
>> (conj '() 1 2 3 4) ;; => (4 3 2 1)
```

Wichtig ist, dass durch diese Operationen auf Listen die Argumente nicht verändert werden, sondern neue Listen erzeugt werden:

```
>> (def Leer '())
>> (conj Leer 3 2 1) ;; => (1 2 3)
>> Leer ;; => ()
```

Listen als Funktionsanwendung

Ohne „'" vor der Liste wird diese auf besondere Weise ausgewertet: Zuerst wird der erste Ausdruck in der Liste ausgewertet. Ist das Ergebnis eine Funktion, werden alle weiteren Ausdrücke der Liste ausgewertet und die Funktion mit diesen Werten als Parameter aufgerufen. Ist das Ergebnis der Auswertung des ersten Ausdrucks keine Funktion (oder Sonderform), tritt ein Fehler auf.

Dem Ausdruck in *1 + 2 * 3* in Infix-Notation entspricht in Clojure >> *(+ 1 (* 2 3)) ;; => (+ 1 6)* => *7*. In diesem Fall ist der erste Ausdruck in der Liste das Symbol +, das standardmäßig an die Additionsfunktion gebunden ist. Als Argumente folgen ein konstanter Ausdruck und wieder ein Funktionsaufruf. Die Prefix-Schreibweise ist vielleicht ungewohnt, hat aber den Vorteil, dass die Operatorreihenfolge eindeutig durch die Klammerung vorgegeben ist.

Funktionen erzeugen

Eine Funktion wird mit *(fn [<argument>*] <ausdruck>*)* erzeugt. Jede Funktion wertet die Ausdrücke im Rumpf nacheinander aus und liefert den Wert des zuletzt ausgewerteten Ausdrucks zurück. Funktionen können mit einer variablen Argumentzahl definiert werden. Dazu ist vor den letzten Parameter der Argumentliste ein „*&*" zu stellen. Der letzte Parameter wird dann an die Liste aller verbleibenden Argumente gebunden: >> *((fn [x & xs] xs) 0 1 2 3 4) ;; => (1 2 3 4)*.

Neben *fn* existiert eine kürzere (und eingeschränktere) Syntax für die Erzeugung von Funktionen. Mit *#(<ausdruck>)* wird eine Funktion erzeugt, die *einen* Ausdruck auswertet, bei dem die Symbole *%1 ... %n* die impliziten Parameter der Funktion sind. Sie sollten diese Syntax nur für kurze Funktionen verwenden.

Funktionen können wie andere Werte mit *def* an Symbole gebunden werden: >> *(def summe (fn [a b] (+ a b)))*. Und mit *defn* existiert eine Kurzform, die beide Schritte vereinigt: >> *(defn summe [a b] (+ a v))*.

In Clojure können Funktionen wie Werte behandelt werden, d. h. sie können an Symbole gebunden (siehe *defn*), als Parameter an Funktionen übergeben und als Ergebnis von Funktionen zurückgegeben werden (siehe *fn*). Das hört sich zunächst unspektakulär an, ermöglicht aber eine elegante Formulierung von Programmen, die ohne dieses Sprachmittel nur umständlich zu erreichen wären.

Der Clojure-Sprachkern im Namensraum *clojure.core* stellt eine große Menge von Funktionen zur Verfügung, die eine Funktion als Parameter erhalten und so in ihrer Funktionsweise angepasst werden können (so genannte „Higher Order Functions": Funktionen höherer Ordnung). *map* ist ein Beispiel für eine HOF. Sie wendet eine Funktion auf alle Elemente einer Collection-Instanz an und liefert eine neue Liste mit den Ergebnissen:

```
>> (def zahlen1-7 (range 1 8)) ;; (range i i+n) => (i,i+1,...,i+n-1)
>> (map #(* %1 %1) zahlen1-7) ;; => (1 4 9 16 25 36 49)
```

map kann auch mit Funktionen angewendet werden, die mehr als ein Argument übernehmen. Für jedes Argument der Funktion *f* wird *map* eine eigene Collection übergeben. Das *i*-te Element des Ergebnisses ist dabei die Anwendung von *f* auf die *i*-ten Elemente der Collections. Die Ergebnismenge ist nur so lang wie die kürzeste Argumentliste: >> *(map list '(1 2 3 4 5) '(:a :b)) ;; => ((1 :a) (2 :b))*.

Ein weiteres Beispiel für eine Higher Order Function aus *clojure.core* ist *reduce*: Es erhält als ersten Parameter eine Funktion *f* mit zwei Argumenten, als zweiten Pa-

rameter einen Wert *r0* und dritten Parameter eine Liste von Werten *e0 ... en*. Die Auswertung geschieht nach:

```
(reduce f r0 '(e0 ... en)) => (f ... (f (f r0 e0) e1) ... en)
>> (reduce conj '() '(1 2 3)) ;; => (conj (conj (conj '() 1) 2) 3) => (3 2 1)
```

Diese Funktion ist in anderen Sprachen auch als „left fold“ bekannt.

Namensräume

Um Namenskollisionen zu vermeiden, ist jede Definition Mitglied in einem Namensraum. Symbole können innerhalb des Namensraums, in dem sie definiert wurden, ohne den vorangestellten Namensraum benutzt werden. Definitionen aus anderen Namensräumen müssen entweder durch Angabe des Namensraums referenziert oder im eigenen Namensraum durch *:refer* oder *:use* bekanntgemacht werden. Der Unterschied von *:use* zu *:refer* ist, dass *:use* eventuell noch nicht geladene Namensräume aus Ressourcen im Klassenpfad von Clojure lädt:

```
>> (ns innoq.a)
>> (def frage "Das Leben, das Universum, alles ...")
>> (def antwort 42)
>> antwort ;; => 42
>> (ns innoq.b)
>> antwort ;; => Unable to resolve symbol: antwort in this context
>> (ns innoq.b (:refer innoq.a :only [antwort]))
>> antwort ;; => 42
>> frage ;; => Unable to resolve symbol: frage in this context.
      ;; Durch ":only [antwort]" im :refer wurden die importierten
      ;; Elemente auf antwort beschränkt.
```

Datenstrukturen

Datenstrukturen sind wichtige Konstrukte einer Programmiersprache. Clojure besitzt neben der schon vorgestellten Liste weitere Datenstrukturen. Eine gemeinsame Besonderheit aller Clojure Collections ist, dass diese unveränderlich sind (sog. persistente Datenstrukturen). Operationen, die z. B. in den analogen Klassen *java.util.** das Objekt ändern, erzeugen in Clojure eine neue Instanz. Clojure Collections können wegen dieser Unveränderbarkeit wie Werte behandelt werden, z. B. bei Vergleichen mit „=“. Clojure nutzt effiziente Implementierung für die persistenten Datenstrukturen, bei denen identische Teile von Strukturen gemeinsam verwendet werden.

Assoziative Strukturen

Eine assoziative Datenstruktur (*Map*) verknüpft beliebige Schlüsselwerte mit beliebigen Werten. Eine Map wird durch die Syntax *{<schlüssel_1> <wert_1> ... <schlüssel_n> <wert_n>}* erzeugt.

Der Zugriff auf ein Element der Map hat die Form einer Funktionsanwendung auf einen Schlüssel, die Map übernimmt die Rolle der Funktion. Optional kann noch ein weiterer Parameter übergeben werden, der als Wert zurückgegeben wird, wenn zum Schlüssel kein Wert in der Map zugeordnet ist:

```
>> (def a-map {:erster 1, :zweiter 2, :dritter 3})
>> (a-map :erster) ;; => 1
>> (a-map :sonstige) ;; => nil
>> (a-map :sonstige 100) ;; => 100
```

Als Beispiel für die Nutzung von Maps dient die Definition eines endlichen Zustandsautomaten. Die Schlüssel sind die Zustände das Automaten, die Werte Maps, die Ereignisse auf Zielzustände abbilden:

```
>> (def fsm {:StateA  {:EventX :StateB, :EventY :StateC, :EventZ :StateA}
           :StateB  {:EventX :StateA, :EventZ :StateC}
           :StateC  {:EventX :StateX, :EventY :StateA}})
```

Die Funktion *fsm-run* liefert für den Startzustand und eine beliebige Zahl von Ereignissen die eingenommenen Zustände, wobei ein ungültiger Übergang zum Zustand *:ILLEGAL* führt:

```
>> (defn fsm-run[state & events]
    (reverse  ;; Liste mit umgekehrter Elementreihenfolge erzeugen
     (reduce (fn [history event]
              (cons ((fsm (first history) {}) event :ÌLLEGAL)
                  history))
            (list state)
            events)))
>> (fsm-run :StateA :EventX :EventY :EventY);;=>
                                    (:StateA :StateB :ÌLLEGAL :ÌLLEGAL)
```

Clojure besitzt eine recht umfangreiche Sammlung von Funktionen für den Umgang mit Map. Hier zwei Beispiele: Durch *(assoc <map> <key_1> <val_1> <key_n> <val_n>)* wird eine Map erzeugt, die *<map>* um die Schlüssel-/Wertpaare *<key_i>/<val_i>* erweitert bzw. schon vorhandene Assoziationen ersetzt. Wichtig: Die ursprüngliche Map wird nicht verändert, sondern eine neue zurückgegeben:

```
>> (assoc fsm :NewState {:EventX :StateA :EventY :StateB})
>> (fsm :NewState) ;; => nil
```

Die Funktion *(zipmap <keys> <values>)* erzeugt aus zwei Collections eine Map, die das n-te Element der *<keys>* mit dem n-ten Element der *<values>* verknüpft. Überzählige Elemente in einer Collection werden ignoriert: *>> (zipmap '(:1ter :2ter :3ter) '(1 2 3 4 5)) => {:3ter 3, :2ter 2, :1ter 1}*.

Vektoren

Ein Clojure-Vektor erlaubt über einen ganzzahligen Index Zugriff auf seine Elemente. Dabei ist die Zugriffszeit (fast) unabhängig von der Vektorlänge. Die konkrete Syntax für die Erzeugung eines Vektors ist *[<element>*]*:

```
>> (def a-vect ["a" "b" "c" "d" ["x" "y"]])
```

Die Syntax für den Zugriff auf ein Vektorelement hat die Form eines Funktionsaufrufs. Tatsächlich kann ein

Clojure-Vektor als eine Funktion aufgefasst werden, die eine natürliche Zahl auf einen Wert abbildet:

```
>> (a-vect 0) ;; => "a"
>> (a-vect 4) ;; => ["x" "y"]
```

Damit ist ein Clojure-Vektor *V* eine spezielle assoziative Datenstruktur, deren erlaubte Schlüsselwerte auf die natürlichen Zahlen aus *[0,...,(count V)-1]* eingeschränkt sind. Wie bei den Maps können auch bei einem Vektor durch *(assoc <vector> <idx_i> <wert_i> ...)* Elemente gesetzt werden. Allerdings sind die Werte von *<idx_i>* auf *[0,...(count <vector>))]* beschränkt. Ist der Wert des Index gleich der Vektorlänge, wird der Wert am Ende des resultierenden Vektors angefügt: >> *(assoc [0 1] 0 :a 1 :b 2 :c) ;; => [:a :b :c]*.

Umgebungen

Wenn innerhalb von Funktionen berechnete Ausdrücke häufiger verwendet werden sollen, ist es nützlich, auf den Wert über einen symbolischen Namen zugreifen zu können. Bisher haben Sie zwei Möglichkeiten kennengelernt, Symbole mit Werten zu verknüpfen: *def*, das ein Symbol global an einen Wert bindet. Die zweite Möglichkeit ist nicht so offensichtlich, es sind Funktionen. Innerhalb einer Funktion sind die Parameter an die Werte des Aufrufs gebunden. *def* hat den Nachteil, dass eine globale Bindung erzeugt wird, die Nutzung von Funktionen ist für den Zweck etwas umständlich. Daher existiert mit *let* eine weitere Form, mit der Sie Symbole und Werte lokal verknüpfen können: *(let [<sym1> <expr1> ... <symn> <exprn>] <body>*)*.

Die Ausdrücke *<expr_i>* werden nacheinander ausgewertet und an die Symbole *<sym_i>* gebunden. In den Ausdrücken *<expr_j>* können die vorhergehenden Symbole mit ihrem Wert aus dem *let* verwendet werden. Für die *<body>*-Ausdrücke innerhalb des *let* sind die Symbole an die Werte gebunden. Rückgabewert des *let*-Ausdrucks ist der letzte ausgewertete *<body>*. Beim Verlassen des *let* gelten die alten Bindungen für die Symbole:

```
>> (def x "ix")
>> (def y "ypsilon")
>> (let [x 2
        y (* x x)]  ;; Hier gilt x=2
    (+ x y)) ;; => 6
>> x ;; => "ix"
>> y ;; => "ypsilon"
```

Auf den ersten Blick hat die *let*-Form Ähnlichkeiten mit der Verwendung von lokalen Variablen in Java oder anderen imperativen Sprachen. Es besteht jedoch der wesentliche Unterschied, dass die innerhalb des *let* erzeugten Bindungen unveränderlich sind.

Kontrollstrukturen

Kontrollstrukturen ermöglichen es, Ausdrücke nur unter bestimmten Bedingungen auszuführen. Die einfachste Form ist *(if <bedingung> <wenn-wahr> <wenn-falsch>?)*. Die Bedingung wird ausgewertet und wenn ihr Wert einem booleschen *true* entspricht, der folgende Ausdruck ausgewertet und als Wert des *if* zurückgegeben. Als boolesches *true* zählen neben *true* alle Clojure-Werte mit Ausnahme von *false* und *nil*. Ist der Wert der Bedingung *false* oder *nil*, wird *<wenn-falsch>* ausgewertet und zurückgegeben. Fehlt *<wenn-falsch>*, so ist der Wert des *if*-Ausdrucks *nil*:

```
(if (> 2 3) :groesser :nichtgroesser) ;; =>  :nichtgroesser
```

Eine weitere, häufig benutzte Bedingung ist *cond*. *(cond <test1> <expr1> ...)* – *cond*-Werte die *<test_i>* der Reihe nach bis zum ersten Ausdruck *<expr_j>* aus der *true* liefert. Der Rückgabewert des *cond*-Ausdrucks ist *<expr_j>*. Ist keine der Bedingungen erfüllt, liefert der *cond*-Ausdruck *nil*. Soll *cond* in jedem Fall einen Wert ungleich *nil* liefern, kann als letztes Test-/Wertpaar ein Keyword und der Default-Wert verwendet werden. Zur Demonstration das Spiel „Papier/Schere/Stein". Die „,“ in den geschachtelten c *cond* dienen nur der besseren Strukturierung. Kommata werden in Clojure ignoriert:

```
(defn papier-schere-stein [a b]
  ;; Eine Partei wird hier leicht benachteiligt.
  (if (= a b) :unentschieden
    (cond (= :papier a) (cond (= :schere b) b, (= :stein b) a, :else a)
          (= :schere a) (cond (= :stein b) b, (= :papier b) a, :else a)
          (= :stein a)  (cond (= :papier b) b, (= :schere b) a, :else a)
          :else b)))
```

if und *cond* sind Beispiele für „Sonderformen". Das sind Sprachelemente, deren Argumente nicht in jedem Fall vor der Ausführung der Definition ausgewertet werden. Das unterscheidet Sonderformen von Funktionen.

Spiel das Lied noch einmal, Sam

Für den Fall, dass ein Schleifenkonstrukt benötigt wird, gibt es die Form *(loop [<sym1> <expr1> ... <symn> <exprn>] <expr>*) loop* ist ein *let* mit einer Erweiterung. Wie *let* erzeugt *loop* lokale Symbol-Wert-Bindungen für die im Rumpf ausgewerteten Ausdrücke. Zusätzlich kann innerhalb der Ausdrücke noch der Ausdruck *(recur <val1> … <valn>)* stehen, wenn dieser der letzte ausgewertete Ausdruck im *loop* ist. Durch *recur* werden die Bindungen der lokalen Symbole *sym1 … symn* mit den übergebenen Werten belegt und die Ausdrücke im Rumpf nochmals ausgeführt:

```
(defn bereich „Erzeugt eine Lister der Zahlen (0...n)"
  [n]
  (loop [i 0     ;; Zähler i mit 0 initialisieren
         r '()]  ;; Resultat r mit leerer Liste initialisieren
    (if (< i n)
      (recur (inc i) (conj r i))  ;; recur ist der letzte ausgewertete Ausdruck
      (reverse r))))
```

Schleifen sind in imperativen Sprachen ein häufig verwendetes Konstrukt. In Clojure werden Schleifen weniger oft benötigt, da mit *map*, *filter* und ähnlichen Funktionen die gleichen Aufgaben in vielen Fällen kürzer ausgedrückt werden können. Für die Funktion *bereich* im Beispiel oben ist z. B. bereits die Funktion *range* im Sprachkern enthalten.

Die Form *recur* kann auch innerhalb von Funktionen angewendet werden, wenn sie der letzte ausgewertete Ausdruck in der Funktion ist. Das Verhalten ist hier ähnlich wie bei *loop*: Die Parameter der Funktion werden an die Werte im *recur* gebunden und der Funktionsrumpf ausgewertet. Als Beispiel die bekannte Fakultät:

```
(defn fak „fak n berechnet die Fakultät."
  ([n] (fak n 1))       ;; Definition von fak für ein Argument
  ([n r] (if (= n 0)    ;; Definition mit zweitem Argument für Teilergebnisse
         r
         (recur (dec n) (* n r))))) ;;
```

Clojure erlaubt normale Rekursion durch Aufruf der Funktion über ihren Namen. Das kann im Gegensatz zu *recur* an beliebigen Stellen geschehen, nicht nur als letzter ausgewerteter Ausdruck. Im Beispiel oben könnte die letzte Zeile auch als *(fac (dec n) (* n r))* geschrieben werden. Der Unterschied zu *recur* besteht darin, dass bei Rekursionsaufruf über den Funktionsnamen der Stack verwendet wird und daher die Rekursionstiefe durch die Stack-Tiefe der VM beschränkt ist. Bei Verwendung von *recur* wird die Rekursion in eine Schleife ohne Verwendung des Stacks umgewandelt, die Rekursionstiefe ist daher unbeschränkt.

Veränderungen mit Atomen und Agenten

Es gibt Probleme, in denen eine einfache Lösung die Veränderung eines Werts erfordert. Für diesen Fall gibt es in Clojure „Atome". Ein Atom ist in Clojure eine Referenz auf einen Wert, die von verschiedenen Threads geändert werden kann. Ein Atom wird mit *(atom <wert>)* angelegt. Der aktuelle Wert eines Atoms kann mit *@<atom>* abgefragt und mit *(swap! <atom> <fun>)* geändert werden. *swap!* wendet die Funktion *<fun>* auf den Wert von *<atom>* an und ersetzt den aktuellen Wert durch das Ergebnis. Die Zuweisung ist Thread-sicher, auch wenn mehrere Threads parallel auf dem gleichen Atom arbeiten, ist der Vorgang atomar:

```
>> (def zahl (atom 2))
>> @zahl ;; => 2
>> (swap! zahl #(* % %)) ;; => 4
>> @zahl ;; => 4
```

Ein Clojure-Atom ermöglicht die synchrone Änderung eines Werts durch verschiedene Threads. Mit Clojure-Agenten existiert eine Möglichkeit, asynchrone Änderungen an einem Zustand durchzuführen. Agenten besitzen einen Zustand, der durch das „Senden" einer Funktion an den Agenten verändert werden kann. Mit *(agent <startwert>)* wird ein Clojure-Agent erzeugt und dessen Anfangszustand mit *<startwert>* belegt, der aktuelle Zustand wird durch *@<agent>* zurückgeliefert und durch *(send <agent> <funktion> <parameter>*)* wird der Zustand des Agenten aktualisiert. Dabei wird *<funktion>* mit dem Agentenzustand als erstem Argument und den weiteren Parametern ausgewertet und das Ergebnis als neuer Zustand gesetzt. Entscheidend ist, dass der *send* aufrufende Thread nicht auf die Auswertung wartet. Die Funktion wird in einem eigenen Thread gestartet. Der Agent, der die Funktion auswertet, ist innerhalb der Funktion an das Symbol **agent** gebunden.

Als Beispiel folgen zwei Agenten, die als Zustand einen Vektor mit dem Namen und einem Zähler halten. Die Funktion *ping-pong* erhält als erstes Argument den Zustand des Agenten, dann den Agenten, an den eine Antwort gesendet werden soll, und dann die verbleibende Anzahl von zu sendenden Nachrichten:

```
>> def a0 (agent [:a0 0]))
>> (def a1 (agent [:a1 0]))
>> (defn ping-pong [state sender n]
     (let [[name counter] state]
       (if (> n 0)
         (do
          (println "Hello " name)
          (send sender ping-pong *agent* (dec n))
          [name (+ 1 counter)])
        state)))
>> (send a0 ping-pong a1 10)
>> @a0 ;; => [:a0 5]
```

Java-Integration

Clojure wurde ausdrücklich als ein Lisp für die Java-VM konzipiert und so ist die Nutzung von Java-Klassen recht problemlos. Alle Klassen, die im Klassenpfad der Java-VM der Clojure-REPL liegen, können im Clojure-Code genutzt werden. Alle Sprachelemente von Clojure sind als Java-Klassen implementiert bzw. werden in Java-Klassen übersetzt. Viele der Clojure-Datentypen sind Standardtypen von Java. Die Java-Klasse eines Clojure-Ausdrucks kann mit der Funktion *class* abgefragt werden:

```
>> (class true) ;; => java.lang.Boolean
>> (class 1)  ;; => java.lang.Integer
>> (class "Text")  ;; => java.lang.String
```

Methoden von Objekten können in Clojure mit *(.<methode> <object> <parameter>*)* aufgerufen werden, Instanzen von Objekten können mit der Form *(<klasse>. <parameter>*)* erzeugt werden. Statische Methoden sind durch *(<klasse>/<methode> <parameter>*)* zugänglich:

```
>> (def java (String. „JAVA"))
>> (.toLowerCase java) ;; => „java"
>> (Math/sin (Math/PI)) ;; => 1.2246467991473532E-16 ;; Fast richtig ☺
```

Zum Abschluss ein Programm, das ein Schiebepuzzle mit einer Swing-GUI implementiert. Das Programm wird in einem eigenen Namensraum implementiert, in dem durch *:import* Java-Klassen durch ihren unqualifizierten Namen genutzt werden können:

```
(ns innoq.schiebung
 [:import [javax.swing JFrame JPanel JButton]]
 [:import [java.awt GridLayout]])
```

Zunächst werden Konstanten und zwei Hilfsfunktionen implementiert. *zeile-spalte* rechnet einen Index aus einem eindimensionalen Array in einen Vektor mit Zeile und Spalte um:

```
(def LAENGE 4)
(def ANZAHL (* LAENGE LAENGE))
(defn zeile-spalte [idx]
    [(int (/ idx LAENGE)) (rem idx LAENGE)])
```

Die Funktion *beweglich?* testest, ob die Indizes *idx* und *frei* direkte Nachbarn in derselben Spalte oder Zeile sind:

```
(defn beweglich? [idx frei]
  (let [[idx-zeile idx-spalte] (zeile-spalte idx)
        [frei-zeile frei-spalte] (zeile-spalte frei)]
    (or (and (=  idx-zeile frei-zeile)
             (or (= frei-spalte (inc idx-spalte))
                 (= frei-spalte (dec idx-spalte))))
        (and (=  idx-spalte frei-spalte)
             (or (= frei-zeile (inc idx-zeile))
                 (= frei-zeile (dec idx-zeile)))))))
```

Die Funktion *action-listener* erzeugt durch die eingebaute Clojure-Funktion *proxy* ein Java-Objekt, das das *ActionListener*-Interface implementiert. Die Werte, die *action-listener* übergeben werden, sind innerhalb des konstruierten Objekts verfügbar, es handelt sich um einen so genannten „Funktionsabschluss“ oder „Closure“:

```
(defn action-listener
  „idx ist der Feldindex des Elements, dem der ActionListener zugeordnet ist,
   frei ist ein Clojure-Atom, das den Index des freien Feldes im Puzzle hält
   buttons ist ein Vektor mit allen Buttons des Puzzles.“
  [idx frei buttons]
  (proxy [java.awt.event.ActionListener][]
    (actionPerformed [e]
      (if (beweglich? idx @frei)  ;; @frei : der Index des freien Feldes
        (let [source (.getSource e)]
          (.setText (buttons @frei) (.getText  source))
          (.setText source "")
          (swap! frei (constantly idx)))))))
```

Zusammengebaut und angezeigt wird alles in der Funktion *schiebepuzzle*. Im *let* wird ein Clojure-Atom angelegt. Dieses dient als Zeiger auf einen veränderlichen Wert. Im Schiebepuzzle wird in diesem Wert der Index des freien Felds abgelegt. Es folgen die GUI-Elemente: der Frame *frame*, ein Panel mit GridLayout und ein Vektor mit Buttons. Den Buttons werden dann ActionListener zugewiesen. Diese haben Zugriff auf die Position des Buttons, dessen Ereignisse sie behandeln, auf das Atom mit der Position des freien Felds sowie auf alle anderen Buttons des Puzzles:

```
(defn schiebepuzzle []
  (let [frei (atom 0)
        frame (JFrame. "Schiebung")
        fpanel (JPanel. (GridLayout. LAENGE LAENGE))
        buttons (vec (doall
                  (map (fn [idx] (JButton. (if (= 0 idx) "" (str idx))))
                       (range ANZAHL))))]
    (dorun (map (fn [btn] (.add fpanel btn)) buttons))
    (loop [i 0, btns buttons]
      (if (not (empty? btns))
        (do (.addActionListener (first btns)
                                (action-listener i frei buttons))
            (recur (inc i) (rest btns)))))
    (.add frame fpanel)
    (.pack frame)
    (.setVisible frame true)))

(schiebepuzzle)
```

Viel Spaß beim Schieben!

Burkhard Neppert ist Senior Consultant bei der innoQ Deutschland GmbH. Er beschäftigt sich dort mit modellgetriebenen Entwicklungsmethoden und dem Entwurf von IT-Systemen.

Links & Literatur

[1] Counterclockwise: http://code.google.com/p/counterclockwise/

[2] http://www.assembla.com/wiki/show/clojure/Getting_Started_with_Emacs

[3] Tolksdorf, R.: „Programming languages for the Java Virtual Machine JVM and JavaScript“: http://www.is-research.de/info/vmlanguages/