Brauer, Johannes; Crasemann, Christoph; Jasser, Stefanie; Krasemann, Hartmut

Working Paper

Eine DOSL für Clojure

Arbeitspapiere der Nordakademie, No. 2015-02

Provided in Cooperation with:
Nordakademie - Hochschule der Wirtschaft, Elmshorn

Suggested Citation: Brauer, Johannes; Crasemann, Christoph; Jasser, Stefanie; Krasemann, Hartmut (2015) : Eine DOSL für Clojure, Arbeitspapiere der Nordakademie, No. 2015-02, Nordakademie - Hochschule der Wirtschaft, Elmshorn

This Version is available at:
https://hdl.handle.net/10419/112723

ARBEITSPAPIERE DER NORDAKADEMIE

ISSN 1860-0360

Nr. 2015-02

Eine DOSL für Clojure

Johannes Brauer, Christoph Crasemann, Stefanie Jasser, Hartmut Krasemann

Juni 2015

Eine elektronische Version dieses Arbeitspapiers ist verfügbar unter
http://www.nordakademie.de/arbeitspapier.html

NORDAKADEMIE
HOCHSCHULE DER WIRTSCHAFT

Köllner Chaussee 11
25337 Elmshorn
http://www.nordakademie.de

Eine DOSL für Clojure

Johannes Brauer, Christoph Crasemann, Stefanie Jasser,
Hartmut Krasemann

1. Juni 2015

Inhaltsverzeichnis

Abstract

Für den Objects-first-Ansatz in der Lehre der objektorientierten Programmierung – ohne den Ballast von Klassen – fehlt es bisher an geeigneten Programmiersprachen. Die DOSL (Direct Object Specification Language) erweitert deshalb bisher Smalltalk um die Möglichkeit, Objekte direkt, d.h. ohne Klassen, zu definieren.

Um den Objects-first-Ansatz auf funktionale Programmiersprachen wie Clojure ausdehnen zu können, wurde eine erste, noch eingeschränkte Version der DOSL in Clojure implementiert. Diese Arbeit beschreibt die Implementierung der DOSL in Clojure im Detail mit einem Ausblick auf die vollständige Version der DOSL.

Während die DOSL in Smalltalk mit Hilfe von Parser-Kombinatoren als Spracherweiterung implementiert wurde, kommt in der Clojure-Variante klassische Lisp Makro Programmierung zum Einsatz.

1 Einleitung

Das Akronym *DOSL* steht für *direct object specificaton language* und benennt eine Sprache, die die Definition von Objekten im Sinne der objektorientierten Programmierung ermöglicht, ohne vorher

eine Klasse angelegt haben zu müssen. Es handelt sich also um eine klassenlose, objektorientierte Sprache. Technisch wird die DOSL unter Einsatz von Clojure-Makros wie eine domänenspezifische Sprache (DSL) realisiert. Gleichwohl ist der Einsatzbereich der DOSL nicht domänenspezifisch sondern stellt eher eine Erweiterung der Ausdrucksmöglichkeiten von Clojure bereit. Beide Varianten der DOSL sind im Kontext didaktischer Fragestellungen in der Programmiergrundausbildung entstanden, die seit geraumer Zeit häufig unter dem Motto „Objects first“ stattfindet, obwohl die dabei verwendeten Programmiersprachen in der Regel klassenbasiert sind und damit eigentlich nur einen „Classes first“-Ansatz erlauben.

Andere Beispiele derartiger Programmiersprachen sind JavaScript (`http://www.ecma-international.org/publications/standards/Ecma-262.htm`) und Self (`http://www.selflanguage.org`).

Die Autoren haben bereits eine ähnliche Sprache als Erweiterung der Syntax von Smalltalk entwickelt. vgl.

Brauer, Johannes ; Krasemann, Hartmut ; Crasemann, Christoph: Implementierung einer in Smalltalk eingebetteten Sprache für die Erzeugung klassenloser Objekte. In: *Forschung für die Wirtschaft 2012*. NORDAKADEMIE Hochschule der Wirtschaft, 2012

Die Programmiersprache Clojure[1] ist ein Lisp-Dialekt, der den funktionalen Programmierstil in den Vordergrund stellt und sich insbesondere durch eine effiziente Implementierung von funktionalen Datenstrukturen[2] auszeichnet. Das bedeutet, dass – wo immer möglich – zustandslosen Berechnungen der Vorzug gegenüber zustandsbehafteten gegeben wird. In der hier beschriebenen DOSL werden daher zunächst Objekte eingeführt, die ihren Zustand nicht ändern können. Eine zweite Version der Sprache[3] wird dann auch die Definition zustandsbehafteter Objekte ermöglichen.

[1] `http://clojure.org`

[2] Der englische Fachbegriff lautet. persistent data structures

„Doing encapsulation right is a commitment not just to abstraction of state, but to eliminate state oriented metaphors from programming.“ Alan Kay, Early H istory of S ma!talk

[3] Diese wird in diesem Aufsatz nicht behandelt.

Eine DOSL mit Clojure als Wirtssprache erlaubt, im Programmierunterricht einerseits einen echten Objects-first-Ansatz zu verfolgen, andererseits aber auch mit einem Einstieg in die funktionale Programmierung zu beginnen. Zustandslose (funktionale) Objekte könnten dann nahtlos zu einem späteren Zeitpunkt eingeführt werden. Die Bildung von Klassen würde dann als expliziter Abstraktionsmechanismus behandelt.

An dieser Stelle kann dann auch grundsätzlich über den Nutzen der Verknüpfung von Struktur und Verhalten diskutiert werden.

2 *Klassen und Objekte in Clojure*

Die Idee, Techniken der objektorientierten Programmierung in einer funktionalen Programmiersprache zu ermöglichen ist nicht neu.[4] Man definiert eine Funktion A, die als Resultat eine Funktion a mit lokalen Variablen für Attribute und Methoden liefert. Funktionen wie A dienen dann als Ersatz für Klassen. Sie liefern einen Methoddispatcher a als Resultat; a ist ein Exemplar von A.

[4] Für Scheme wird das z. B. hier erläutert: .

Abelson, Harold ; Sussman, Gerald J. ; Sussman, Julie: *Structure and interpretation of computer programs*. The MIT Press, 1999

2.1 *Objekte als Closures*

Die Definition von Klassen als Funktionen wird anhand eines einfachen Beispiels in Clojure erläutert. Eine Klasse `person` soll die Exemplarvariablen `name` und `vorname` sowie die dazugehören Getter und Setter besitzen. Außerdem soll eine Funktion (Methode) `vollname` eine Zeichenkette aus Vor- und Nachname der Person liefern.

Die Klassendefinition könnte dann wie folgt aussehen:

```
1  (defn person []
2    (let [name (atom nil)
3          vorname (atom nil)
4          set-name (fn [n] (reset! name n))
5          set-vorname (fn [n] (reset! vorname n))
6          get-name (fn [] @name)
7          get-vorname (fn [] @vorname)
8          vollname (fn [] (str @vorname " " @name))]
9      (fn [message]
10       (cond (= message :set-name) set-name
11             (= message :set-vorname) set-vorname
12             (= message :get-name) get-name
13             (= message :get-vorname) get-vorname
14             (= message :get-vollname) vollname
15             :else (throw (Exception.
16                    (str "Message not understood: "
17                         message)))))))
```

Die Exemplarvariablen `name` und `vorname` werden hier als Clojure-Atoms (`http://clojure.org/atoms`) definiert. Atoms stellen eine (von mehreren) Möglichkeiten dar, zustandsbehaftete Variablen zu verwalten. Die `reset!`-Funktion realisiert die Zuweisung. Die Schreibweise `@name` ist syntaktischer Zucker für `(deref name)` und ermöglicht den Zugriff auf den Wert eines Atoms.

Ein Aufruf der Funktion `person`, z. B. durch

```
(def p (person))
```

erzeugt ein `person`-Objekt, indem er als Resultat die in den Zeilen 9 bis 17 definierte lokale Funktion liefert. Diese bildet eine Closure mit den im `let`-Block definierten lokalen Variablen. Diese Dispatcher-Funktion akzeptiert als Argument für den Parameter `message` ein Clojure-keyword[5]. Damit kann der durch die Variable `p` definierten Person z. B. ein Vorname gegeben werden, indem die zugehörige Set-Funktion im Message-passing-style aufgerufen wird.

[5] Wortsymbole mit vorangestelltem Doppelpunkt, vgl. `http://clojure.org/data_structures`.

```
((p :set-vorname) "Gustav")
```

Die Nachricht wird hier durch das passende Keyword selektiert.

Der Ausdruck `((p get-vorname))` liefert dann die Zeichenkette `"Gustav"`. Nach

```
((p :set-name) "Gans")
```

kann auch die Methode `:vollname` benutzt werden:

```
((p :vollname))   ;; => "Gustav Gans"
```

Zwei Prinzipien der objektorientierten Programmierung, *Geheimnis-* und *Klassenprinzip*, sind durch diese Konstruktion bereits gegeben. Da die Methoden lokal definierte Funktionen sind, ist *Polymorphie* leicht herstellbar. Auf die Darstellung der Realisierung von *Vererbung* soll an dieser Stelle verzichtet werden.

Für die Implementierung von Klassen gibt es noch eine Clojure-typische Variante, bei der anstelle einer Dispatcher-Funktion eine Clojure-Map als Resultat des Aufrufs der Konstruktorfunktion geliefert wird:

```
(defn person []
  (let [name (atom nil)
        vorname (atom nil)
        set-name
        (fn [n]
          (reset! name n))
        set-vorname
        (fn [n]
          (reset! vorname n))
        get-name (fn [] @name)
        get-vorname (fn [] @vorname)
        vollname (fn [] (str @vorname " " @name))]
    {:set-name set-name
     :set-vorname set-vorname
     :get-name get-name
     :get-vorname get-vorname
     :vollname vollname}))
```

Bei dieser Variante wird davon Gebrauch gemacht, dass auf eine Clojure-Map mit einem Schlüssel als Funktionsbezeichner zugegriffen werden kann. So liefert z. B. der Ausdruck `(:b {:a 1 :b 2})` den Wert 2.

Die Methoden können damit auf die gleiche Weise, wie oben dargestellt, aktiviert werden.

2.2 *Funktionale Objekte*

Unter funktionalen Objekten seien hier Objekte verstanden, die ihren Zustand nicht verändern können. Der Zustand wird bei der Erzeugung bestimmt. Überträgt man das Beispiel aus Abschnitt 2.1, so muss die Konstruktorfunktion mit Parametern für die Belegung der Zustandsgrößen bzw. Exemplarvariablen versehen werden, da die Set-Methoden entfallen:

Den Begriff *functional objects* prägte Mattias Felleisen in einer Keynote der ECOOP 2004 in Oslo am 16 Juni 2004

```
(defn person [n v]
  (let [name n
        vorname v
        get-name (fn [] name)
        get-vorname (fn [] vorname)
        vollname (fn [] (str vorname " " name))
        gruss (fn [grussformel] (str grussformel " "
                                     (vollname)))]
    {:get-name get-name
     :get-vorname get-vorname
     :vollname vollname
     :gruss gruss}))
```

Da die Exemplarvariablen nachträglich nicht mehr verändert werden können, werden sie hier nicht mehr als Atoms definiert.

Die Konstruktorfunktion würde jetzt so benutzt:

```
(def p (person "Gans" "Gustav"))
```

Der Ausdruck

```
((p :gruss) "Guten Tag")
```

liefert dann die Zeichenkette

```
"Guten Tag Gustav Gans".
```

Derartige funktionale Objekte stellen das grundlegende Konzept der in Abschnitt 2.3 eingeführten DOSL dar. Diese verzichtet zum Einen auf die Notwendigkeit, eine Klasse[6] angeben zu müssen, zum Anderen braucht kein „boilerplate code“ für die Getter und Setter geschrieben zu werden. Schließlich wird die Syntax des Methodenaufruf vereinfacht.

[6] bzw. eine Konstruktorfunktion

2.3 *Die Clojure-DOSL*

In der Syntax der DOSL wird das `person`-Objekt aus Abschnitt 2.1 wie folgt aufgeschrieben, ohne dass vorher eine Klassendefintion angelegt werden muss:

```
(obj [name "Gans" vorname "Gustav"])
```

Die Namen der Exemplarvariablen und ihre Werte werden in Form eines Clojure-Vectors angegeben; vgl. `http://clojure.org/data_structures`.

Technisch liefert der Ausdruck wieder eine Dispatcher-Funktion, die mit dem Namen einer Methode als Argument aufgerufen werden kann:

```
((obj [name "Gans" vorname "Gustav"]) 'name)
;; => "Gans"
```

Die Get-Methoden werden durch das Clojure-Makro `obj` automatisch angelegt und sind durch Clojure-Symbole[7] zu benennen. Selbstverständlich kann ein DOSL-Objekt auch an ein Symbol gebunden werden:

[7] z. B. `'name`

```
(def p (obj [name "Gans" vorname "Gustav"]))
(p 'vorname)   ;; => "Gustav"
```

Um dies ein wenig abkürzen zu können, gibt es noch das Makro `defobj`:

```
(defobj p [name "Gans" vorname "Gustav"])
```

Jedes DOSL-Objekt versteht die Nachricht `'self`:

```
(p 'self)
;; => (doslclj.core/obj [name "Gans"
                         vorname "Gustav"])
```

`doslclj.core` ist der Clojure-Namespace (vgl. `http://clojure.org/namespaces`), in dem das Makro `obj` definiert ist.

Das Resultat ist der Clojure-Ausdruck, mit dem das Objekt definiert wird.

Bei der Definition von Objekten können neben Exemplarvariablen auch Methoden hinzugefügt werden, die in ihrer Syntax der normaler Funktionen in Clojure folgen:

```
(defobj p [name "Gans"
           vorname "Gustav"
           vollname (fn [] (str vorname " "
                                name))])
(p 'vollname)    ;; => "Gustav Gans"
```

Die DOSL erlaubt auch das nachträgliche Hinzufügen von Slots[8]:

[8] Der Begriff *Slot* wird hier als Oberbegriff für *Exemplarvariable* und *Methode* gebraucht

```
(def p
  (add-slot p gruss
```

```
      (fn [grussformel] (str grussformel " "
                              (vollname)))))
(p 'gruss "Hallo")    ;; => "Hallo Gustav Gans"
```

Falls eine Methode Argumente erwartet, werden diese einfach hinter das Nachrichtensymbol geschrieben.

Das Makro `add-slot` erwartet als Argumente

- ein DOSL-Objekt (hier `p`),
- den Namen der Methode (hier `gruss`) und
- den Lambda-Ausdruck mit der Berechnungsvorschrift.

Zu beachten ist, dass das `add-slot`-Makro immer ein neues DOSL-Objekt liefert. Wertet man nach den obigen Zeilen den Ausdruck

```
(p 'self)
```

aus, erhält man

```
(doslclj.core/obj
 [name
  "Gans"
  vorname
  "Gustav"
  vollname
  (fn [] (str vorname " " name))
  gruss
  (fn [grussformel] (str grussformel " "
                         (vollname)))])
```

als Ergebnis. D. h. die Methode `gruss` ist Bestandteil des Objekts `p`, aber nur, weil das mit `add-slot` erzeugte Objekt mit `(def p ...)` wieder an den Bezeichner `p` gebunden wurde.

Ein Aufruf von add-slot allein

```
(add-slot p gruss
    (fn [grussformel] (str grussformel " "
                           (vollname))))
(p 'gruss "Hallo")
;; =>  Message not understood: gruss
```

ließe das an `p` gebundene Objekt unverändert, was die Nachricht `gruss` nicht versteht. Man kann aber an das von `add-slot` erzeugte Objekt die Nachricht `gruss` direkt senden:

```
((add-slot p gruss
    (fn [grussformel] (str grussformel " "
                           (vollname))))
 'gruss "Guten Tag")
;; => "Guten Tag Gustav Gans"
```

Das Verändern der Werte von Exemplarvariablen ist bei funktionalen Objekten nicht möglich. Dennoch werden bei DOSL-Objekten die Set-Methoden automatisch erzeugt. Ihre Anwendung führt aber immer zur Erzeugung eines neuen Objekts:

```
((p 'name "Ente") 'name)   ;; => "Ente"
(p 'name)                  ;; => "Gans"
```

Ein Setter wird wie ein Getter gefolgt von dem Argument aufgerufen.

Das an `p` gebundene Objekt bleibt unverändert.

3 *Implementierung*

Die Implementierung der DOSL besteht hauptsächlich aus drei Clojure-Makros[9]. Das Makro `obj` spielt dabei die Hauptrolle, da mit seiner Hilfe die DOSL-Objekte erzeugt werden. Das Makro `defobj` stellt lediglich – wie bereits in Abschnitt 2.3 erwähnt – eine vereinfachte Syntax für den Fall bereit, dass ein DOSL-Objekt an ein Clojure-Symbol gebunden werden soll.

Das Makro `add-slot`[10] ermöglicht, aus einem bereits existierenden Objekt ein neues Objekt durch Hinzufügen eines Slosts zu erzeugen.

[9] s. `http://clojure.org/macros`; weitergehende Informationen zu Clojure-Makros ins u. a. hier zu finden:
JONES, Colin: *Mastering Clojure Macros*. The Pragmatic Bookshelf, 2014; and EMERICK, Chas ; CARPER, Brian ; GRAND, Christophe: *Clojure Programming*. OâĂŹReilly Media, Inc, 2012

[10] vgl. Abschnitt 2.3

3.1 *Die Makros `obj` und `defobj`*

Um die Arbeitsweise des `obj`-Makros zu erläutern, soll hier zunächst gezeigt werden, was der Clojure-Makroexpander aus dem Aufruf eines `obj`-Makros macht. Betrachten wir dazu ein einfaches Beispiel. Der Aufruf

```
(macroexpand-1 '(obj [name "Gans" vorname "Gustav"]))
```

liefert als Resultat den folgende Clojure-Ausdruck:

```
1  (clojure.core/let
2   [name "Gans" vorname "Gustav"]
3   (clojure.core/let
4    [slts__3192__auto__
5     (doslclj.core/caller-locals->hashmap)]
6    (clojure.core/letfn
7     [(self__3193__auto__
8       [message__3194__auto__ & args__3195__auto__]
9       (clojure.core/let
10       [value__3196__auto__
11        (clojure.core/cond
12         (clojure.core/= message__3194__auto__ 'self)
13         '(obj [name "Gans" vorname "Gustav"])
14         (clojure.core/contains? slts__3192__auto__
15                                 message__3194__auto__)
16         (message__3194__auto__ slts__3192__auto__)
17         :else
18         (throw
19          (java.lang.Exception.
20           (clojure.core/str
21            "Message not understood: "
22            message__3194__auto__))))]
23       (if
24        (clojure.core/fn? value__3196__auto__)
25        (clojure.core/apply value__3196__auto__
26                            args__3195__auto__)
27        (if
28         (clojure.core/empty? args__3195__auto__)
```

Postfixe der Form `__uvwx__auto__` werden vom Makro-Prozessor an ein Symbol, z. B. `slts` oder `self` angehängt, um Namenskollisionen mit Symbolen der umschließenden Umgebung auszuschließen.

```
29          value__3196__auto__
30          (clojure.core/eval
31           (doslclj.core/new-obj
32            message__3194__auto__
33            (clojure.core/first args__3195__auto__)
34            '(obj [name "Gans" vorname "Gustav"])))))))]
35    self__3193__auto__)))
```

An den Zeilen 1 und 2 ist erkennbar, dass die Makroexpansion einen `let`-Block[11] mit den lokalen Variablen `name` und `vorname` erzeugt. Das Resultat dieses `let`-Blocks ist in der letzten Zeile (35) zu sehen, das Symbol `self__3193__auto__`. An dieses Symbol wird die durch den in Zeile 7 beginnenden `letfn`-Block definierte Funktion gebunden, die in ihrer Funktionsweise der in Abschnitt 2.1 erläuterten Dispatcher-Funktion entspricht. Sie akzeptiert folgende Parameter:

[11] also eine Closure

message Hier wird das die Nachricht identifizierende Symbol als Argument erwartet.

args Dies ist ein optionaler Parameter. Er enthält ggf. eine Liste der Argumente, die von der durch `message` aktivierten Methode erwartet werden.

In dem umschließenden `let`-Block – beginnend mit Zeile 3 – wird der Vektor mit den aus Exemplarvariablen und ihren Werten bestehenden Paaren unter Verwendung der Hilfsfunktion `caller-locals->hashmap` in eine Map verwandelt und an das Symbol `slts` gebunden. Die vom Makroprozessor erzeugten Postfixe der Form `__uvwx__auto__` werden hier und im Folgenden weggelassen.

In Zeile 12 prüft die Dispatcher-Funktion, ob die Nachricht `'self` gesendet wurde und gibt ggf. den Makro-Aufruf (Zeile 13) als Resultat zurück.

In den Zeilen 14 bis 22 wird geprüft, ob es für die gesendete Nachricht eine zugehörige Methode gibt. Sollte das nicht der Fall sein, wird eine Exception[12] ausgelöst.

[12] Zeilen 19 bis 22

Die Zeilen 23 bis 34 behandeln die verschiedenen Formen der Nachrichtensendungen:

- Falls es sich um die Anwendung einer selbst definierten Funktion[13] handelt, wird diese auf ggf. vorhandene Argumente angewendet (Zeilen 25, 26).
- Wenn es keine selbstdefinierte Funktion ist, werden in den Zeilen 27 bis 34 die Getter und Setter unterschieden:
  - Im Falle eines Getters muss nur der an das Symbol `value` gebundene Wert zurückgegeben werden (Zeile 29).
  - Im Falle eines Setters wird in den Zeilen 31 bis 34 ein neues Objekt erzeugt.

  Der Quelltext des Makros `obj` ist in Anhang A angegeben.

[13] z. B. `vollname`, vgl. Abschnitt 2.3

3.2 *Das Makros* `add-slot`

Ohne auf die Implementierung des Makros detailliert einzugehen, soll hier erneut seine Wirkungsweise durch Betrachtung des Resultats der Makroexpansion erläutert werden. Dazu betrachten wir noch einmal das in Abschnitt 2.3 gezeigte DOSL-Objekt

Der Quelltext des Makros `add-slot` ist in Anhang B angegeben.

```
(defobj p [name "Gans"
           vorname "Gustav"
           vollname (fn [] (str vorname " "
                                name))])
```

```
(p 'vollname)    ;; => "Gustav Gans"
```

und die Anwendung des Makros `add-slot`:

```
(add-slot p
   gruss (fn [grussformel] (str grussformel " "
                                (vollname))))
```

Die Makroexpansion resultiert in der Erzeugung eines Aufrufs des Makros `obj`:

```
(doslclj.core/obj
 [name
  "Gans"
  vorname
  "Gustav"
  vollname
  (fn [] (str vorname " " name))
  gruss
  (fn [grussformel] (str grussformel " " (vollname)))])
```

Dieser Makroaufruf entspricht dem für die Definition von `p` zu Beginn dieses Abschnitts, ergänzt um die Definition der Funktion `gruss`. Daraus ist ersichtlich, dass die Anwendung von `add-slot` immer ein neues DOSL-Objekt erzeugt.

4 *Zusammenfassung und Ausblick*

Für den aktuellen Stand der Implementierung sind bisher lediglich ungefähr 60 Zeilen Clojure-Code aufgewendet worden. Als Erweiterung der bisher realisierten Funktionalität werden

- die Ableitung von Klassen aus DOSL-Objekten als expliziter Abstraktionsschritt und
- Objekte mit veränderlichen Zuständen als Alternative zu funktionalen Objekten

ins Auge gefasst.

Mit diesen Erweiterungen könnte ein didaktisches Konzept für die Programmiergrundausbildung im ersten Studienjahr realisiert werden, dass folgendermaßen skizziert werden könnte:

1. Der Einstieg in die Programmierung erfolgt im funktionalen Stil.
2. Der Objektbegriff wird anschließend anhand von funktionalen Objekten eingeführt.
3. Zur Vorbereitung auf die Unterrichtung in herkömmlichen, klassenbasierten, objektorientierten Sprachen kann schließlich der Klassenmechanismus der DOSL benutzt werden.

Die Funktionalitäten der hier vorgestellten DOSL und die didaktische Vorgehensweise können natürlich auch mit einer anderen geeigneten funktionalen Programmiersprache[14] umgesetzt werden.

[14] z. B. Haskell

Der Nutzen der dem objektorientierten Paradigma eigenen Verknüpfung von Struktur und Verhalten in Objekten wird nicht nur

von Anhängern der funktionalen Programmierung infrage gestellt. Auch die Urheber des Data-Context-Interaction-Architekturmusters[15] sehen Objekte eher als Datenbehälter, denen durch die Anwendungslogik verschiedene Rollen zugewiesen werden können. Ob das DCI-Muster mithilfe der Clojure-DOSL adäquat umgesetzt werden kann, ist eine aus der Sicht der Autoren interessante Forschungsfrage.

[15] COPLIEN, James O. ; REENSKAUG, Trygve Mikkjel H.: The data, context and interaction paradigm. In: *SPLASH '12: Proceedings of the 3rd annual conference on Systems, programming, and applications: software for humanity*. New York, NY, USA : ACM, 2012. – ISBN 978–1–4503–1563–0, S. 227–228

A Makros `obj` und `defobj`

Neben den beiden Makros `obj` und `defobj` werden im Folgenden noch zwei Hilfsfunktionen (`duplicated-slot-names` und `new-obj`) und das Hilfsmakro `caller-locals->hashmap` angegeben.

```
(defn duplicated-slot-names [vec]
  (let [slotnames (take-nth 2 vec)]
    (not= (count (into #{} slotnames))
          (count slotnames))))

(defn new-obj [message arg form]
  ;; ersetzt in der obj-Form form den Wert des Slots message
  ;; durch arg und liefert neue obj-Form
  (list 'obj
        (->> (assoc (apply hash-map (second form)) message arg)
             (vec)
             (apply concat)
             (vec))))

(defmacro caller-locals->hashmap []
  (->> (keys &env)
       (map (fn [k] [`(quote ~k) k])) (into {})))

(defmacro obj
  [slots]
  (if (duplicated-slot-names slots)
      (throw (Exception. "duplicated slotname"))
      (let [s# 'self form &form]
        `(let ~slots
           (let [slts# (caller-locals->hashmap)]
             (letfn
                 [(self# [message# & args#]
                    (let [value#
                          (cond
                            (= message# '~s#) '~form
                            (contains? slts# message#) (message# slts#)
                            :else (throw (Exception.
                                          (str "Message not understood: "
                                               message#))))]
                      (if (fn? value#) (apply value# args#)
                          (if (empty? args#)
                            value#
                            (eval (new-obj message#
                                           (first args#) '~form))))))]
               self#))))))

(defmacro defobj
  [sym slots]
  `(def ~sym (obj ~slots)))
```

B Makro `add-slot`

```
defmacro add-slot
  [object methodname function]
  (let [s# 'self
        o# `(~object '~s#)
        slots# `(second ~o#)
        newslots# `(conj (conj ~slots# '~methodname) '~function)]
    (let [slots# (eval newslots#)]
      `(obj ~slots#))))
```

Literatur

[1] ABELSON, Harold ; SUSSMAN, Gerald J. ; SUSSMAN, Julie: *Structure and interpretation of computer programs*. The MIT Press, 1999

[2] BRAUER, Johannes ; KRASEMANN, Hartmut ; CRASEMANN, Christoph: Implementierung einer in Smalltalk eingebetteten Sprache für die Erzeugung klassenloser Objekte. In: *Forschung für die Wirtschaft 2012*. NORDAKADEMIE Hochschule der Wirtschaft, 2012

[3] COPLIEN, James O. ; REENSKAUG, Trygve Mikkjel H.: The data, context and interaction paradigm. In: *SPLASH '12: Proceedings of the 3rd annual conference on Systems, programming, and applications: software for humanity*. New York, NY, USA : ACM, 2012. – ISBN 978–1–4503–1563–0, S. 227–228

[4] EMERICK, Chas ; CARPER, Brian ; GRAND, Christophe: *Clojure Programming*. O'Reilly Media, Inc, 2012

[5] HARTMUT KRASEMANN, Johannes B. ; CRASEMANN, Christoph: Objekte statt Klassen zuerst. In: *Forschung für die Wirtschaft 2011*. NORDAKADEMIE Hochschule der Wirtschaft, 2011

[6] JONES, Colin: *Mastering Clojure Macros*. The Pragmatic Bookshelf, 2014