Goodbye CRUD, hello Immutability: der Umgang mit Daten in Clojure

Michael Sperber, Active Group GmbH

In der rein funktionalen Programmierung sind Daten unveränderlich – also kein gekapselter Zustand, keine Setter, kein CRUD-Pattern. Dieses Paradigma setzt Clojure auf der Java-Plattform konsequent um. Auf den ersten Blick scheint es so, als bedeute der Verzicht auf Veränderung primär eine Einschränkung. Tatsächlich aber befreit gerade dieser Verzicht die Programmierung von vielen Problemen, schafft neue Möglichkeiten und eröffnet eine andere Sicht auf die Datenmodellierung.

Nahezu jedes Standardbeispiel für die objektorientierte Programmierung setzt auf zwei Ideen:

- Ein Objekt repräsentiert eine Entität aus der Problemstellung
- Ein Objekt kapselt unveränderlichen Zustand

In einem Telefonbuch beispielsweise könnten „Personen" gespeichert sein. Entsprechend diesen Ideen gibt es eine Klasse, die in Java anfangen könnte wie in *Listing 1*. Der Bezug zwischen der Idee „Person" aus der Problemstellung und den Attributen der Klassen-Definition ist klar ersichtlich. Außerdem werden die Attribute durch Veränderung manipuliert: Es gibt einen Setter für „firstName", wahrscheinlich auch noch einen für „lastName". An „phones" wird zwar nicht zugewiesen, stattdessen sorgt die Methode „put" unmittelbar dort für eine Veränderung. So ist die Klasse allerdings noch nicht fertig; sie sollte noch eine sinnvolle Definition für „equals" enthalten *(siehe Listing 2)*.

Diese Definition offenbart zwei Probleme:

- Sie ist umständlich, weil die Klasse das CRUD-Pattern benutzt und es damit sein kann, dass „firstName" und „lastName" jeweils „null" sein können.
- Sie ist komplett mechanisch – wahrscheinlich wurde sie von einer IDE generiert. Warum muss sie überhaupt programmiert werden?

Außerdem fehlen noch Definitionen für „hashCode" und „toString", die beide ähnliche Symptome aufweisen. Die Definition weist zudem auf ein anderes Problem hin: Sie implementiert die Idee, dass zwei Personen gleich sind, wenn sie gleiche Namen und gleiche Telefonnummern haben *(siehe Listing 3)*.

Der Vergleich „p1.equals(p2)" liefert „true". Was ist aber nach der Anweisung „p1.setLastName("Ferber");"? Vielleicht hat „p1" geheiratet? Plötzlich sind „p1" und „p2" verschieden: „p2" hat nicht geheiratet; „p1" und „p2" sind nicht dieselbe Person. Ob „das Gleiche" oder „dasselbe" die richtige Grundlage für die Implementierung von „equals" ist, ist Ermessensfrage und manchmal gar nicht so einfach zu entscheiden.

Joshua Blochs Klassiker „Effective Java“ *[1]* hat dementsprechend eine klare Empfehlung, nämlich sogenannte „value objects“ zu verwenden, die unveränderlich („immutable“) sind: „Classes should be immutable unless there's a very good reason to make them mutable … If a class cannot be made immutable, limit its mutability as much as possible.“ Dieser Ratschlag ist in der Java-Welt aus einer Reihe von pragmatischen Gründen weithin akzeptiert:

- „Value objects“ sind problemlos als Schlüssel in Maps und anderen Datenstrukturen verwendbar
- „Value objects“ sind „thread“-sicher
- „Value objects“ benötigen keinen Copy-Konstruktor und auch keine „clone“-Methode

Warum Schlüssel in Maps? Bei Hash-Maps beispielsweise ändert sich der Hash-Code, wenn der Schlüssel verändert wird – dieser steht dann plötzlich nicht mehr an der richtigen Stelle. Dies sind alles gute Gründe. Sie verstellen jedoch den Blick auf die fundamentale Ursache der Probleme, die durch veränderliche Objekte verursacht werden; die Welt funktioniert nämlich nicht so, wie die objektorientierte Programmierung sie mit veränderlichen Objekten modellieren möchte.

Dies erscheint erst einmal nicht intuitiv. In der realen Welt ändert sich alles ständig – die Welt und die Objekte in ihr haben einen Zustand. Allerdings suggeriert das objektorientierte Modell des gekapselten Zustands, dass jedes Objekt seinen eigenen isolierten Zustand hat. Hier ein einfaches Beispiel: Ein Elefant geht von einem Zimmer in den Flur. Ein typisches objektorientiertes Programm macht in dem Beispiel drei Entitäten aus (Elefant, Zimmer, Flur) und aus dem Vorgang möglicherweise diese Folge von Anweisungen: „room1.exit(elephant)“ und „hallway.enter(elephant)“.

Das sind also zwei Schritte; in der Realität braucht der Elefant allerdings nur einen Schritt für beides: Der Schritt aus dem Zimmer ist derselbe Schritt wie der Schritt in den Flur. Das mag als Spitzfindigkeit erscheinen, ist es jedoch spätestens dann nicht mehr, wenn das Programm nebenläufig wird. Das muss es, wenn es die hochgradig nebenläufige Realität modellieren möchte. Dann können andere Threads den Zustand zwischen den beiden Anweisungen beobachten, und da ist der Elefant kurz mal nirgendwo: Der Zustand ist inkonsistent. Es ist möglich, das objektorientierte Modell zu ändern, aber das fundamentale Problem bleibt.

Dass also die reale Welt nicht mit gekapseltem Zustand funktioniert, ist eine Ursache des Problems. Noch fundamentaler ist der Umstand, dass Computerprogramme von Menschen geschrieben werden. Die Objekte in einem Computerprogramm repräsentieren deshalb nicht die Welt, sondern unsere Wahrnehmung der Welt. Dies ist eine der Motivationen für Domain-driven Design.

Auch dazu ein Beispiel: Bei einem Bundesligaspiel gibt es enorm viele Entitäten: 22 Spieler/innen, Schiedsrichter, den Ball, Tausende von Zuschauern. Ein Zuschauer nimmt den Zustand des Spiels immer konsistent wahr; in der visuellen Wahrnehmung befinden sich niemals zwei Personen gleichzeitig am selben Ort oder verschwinden kurz, um dann woanders wieder aufzutauchen. Außerdem funktioniert der Prozess der Wahrnehmung selbst so, dass wir konsistente Schnappschüsse aufzeichnen, die das Gehirn dann analysiert.

```
public class Person {
    private String firstName;
    private String lastName;
    private Map<String, String> phones =
      new HashMap<String, String>();
    public String getFirstName() {
        return firstName;
    }
    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }
    public void newPhone(String key, String to) {
        this.phones.put(key, to);
    }
   ...
}
```

Listing 1

```
@Override
    public boolean equals(Object o) {
        if (this == o)
            return true;
        if (!(o instanceof Person))
            return false;

        Person person = (Person) o;

        if (firstName != null
            ? !firstName.equals(person.firstName)
            : person.firstName != null)
            return false;
        if (lastName != null
            ? !lastName.equals(person.lastName)
            : person.lastName != null)
            return false;
        if (!phones.equals(person.phones))
            return false;
        return true;
    }
```

Listing 2

```
Person p1 = new Person();
p1.setFirstName("Mike");
p1.setLastName("Sperber");
p1.addPhone(new Phone("555"));
Person p2 = new Person();
p2.setFirstName("Mike");
p2.setLastName("Sperber");
p2.newPhone("work", "(123) 555-1234"));
```

Listing 3

```
(ns phonebook
  (:require [active.clojure.record :refer :all]))
```

Listing 4

```
(define-record-type Person
  (make-person first last phones)
  person?
  [first person-get-first
   last person-get-last
   phones person-get-phones])
```

Listing 5

Dies steht in Gegensatz zum Standardmodell der objektorientierten Programmierung für die Wahrnehmung von Ereignissen, dem Observer-Pattern: Ein Zuschauer schickt nicht jedem Objekt im Stadion eine Nachricht, die darum bittet, über Zustandsänderungen informiert zu werden. Außerdem sind die Schnappschüsse häufig noch präsent, wenn die Welt sich weiterbewegt hat. Der Mensch hat ein Gedächtnis und es ist oft enorm nützlich, sich an die Vergangenheit zu erinnern.

Das Problem mit veränderlichen Daten war schon früh bekannt. Alan Kay, der den Begriff „objektorientiert" geprägt hat, schrieb im Jahr 1993 *[2]*: „Though OOP came from many motivations, two were central. ... [T]he small scale one was to find a more flexible version of assignment, and then to try to eliminate it altogether." Irgendwie ist dieses Ziel in der Weiterentwicklung von OOP seit dieser Zeit abhandengekommen.

Daten in Clojure

Der vorige Abschnitt hat gezeigt, warum es sinnvoll ist, auf Veränderlichkeit von Daten zu verzichten. Wie machen die funktionalen Programmierer das, ohne ständig das Gefühl zu haben, dass ihnen etwas Wichtiges fehlt? Die folgenden Code-Beispiele verwenden die Library Active Clojure *[3]*. An den Anfang gehört folgende Präambel für die entsprechenden Importe *(siehe Listing 4)*. Das „ns" steht für „namespace", den Clojure-Namen für „package". *Listing 5* zeigt das Clojure-Pendant zur „Person"-Klassendefinition von oben.

„Record" ist das Clojure-Pendant zu Java-„Pojos" – zusammengesetzten Objekten, deren Bestandteile Namen haben. Clojure hat auch noch ein anderes eingebautes Konstrukt zur Deklaration von Records („defrecord), das sich aber nicht so gut für Erläuterungen eignet. Diese Deklaration definiert den Typ für Personen, außerdem einen Konstruktor „make-person" sowie „Getter"-Funktionen für die Felder „first", „last" und „phones" namens „person-get-first", „person-get-last" und „person-get-phones". Benutzen lässt sich das wie in *Listing 6*.

Hier sind zwei Variablen „mike-sperber" und „sabine-ferber" deklariert – der Konstruktor „make-person" akzeptiert für jedes Feld ein Argument. Die Syntax „{...}" erzeugt eine Map, in dem Fall mit den Schlüsseln „:home" und „:work". Dabei handelt es sich um sogenannte „Keywords", ein spezielles Clojure-Konstrukt, das besonders schnellen Map-Zugriff erlaubt. Das Definieren von „equals" und „hashCode" (oder gar eines Copy-Konstruktors oder „clone") ist unnötig, da Records unveränderlich sind. Die Definitionen werden von Clojure automatisch im Hintergrund erzeugt. Die Getter sind ganz normale Funktionen *(siehe Listing 7)*.

Angenommen, die beiden heiraten. Das Verändern der Felder ist nicht möglich. Stattdessen muss ein neues „Person"-Objekt erzeugt werden. Die Funktion in *Listing 8* erledigt das. Die Funktion „marry" akzeptiert Argumente für die zwei Parameter „person-1" sowie „person-2" und übernimmt (etwas willkürlich) für das neue Objekt den Vornamen von „person-1", den Nachnamen von „person-2" und die Telefonnummern von „person-1". Clojure druckt das Ergebnis wie in *Listing 9* aus. Das Beispiel zeigt, dass ein „Person"-Objekt nicht „eine Person" repräsentiert, sondern den Zustand (beziehungsweise einen Schnappschuss des Zustands) zu einem bestimmten Zeitpunkt. Tritt ein neuer Zustand ein, muss ein neues

```
(def mike-sperber
  (make-person "Mike" "Sperber"
               {:home "(123) 555-1234"
                :work "(321) 555-4321"}))

(def sabine-ferber
  (make-person "Sabine" "Ferber"
               {:home "(123) 555-1234"
                :work "(444) 555-4444"}))
```

Listing 6

```
(person-first mike-sperber) => "Mike"
(person-last sabine-ferber) => "Ferber"
```

Listing 7

```
(defn marry [person-1 person-2]
  (make-person (person-get-first person-1)
               (person-get-last person-2)
               (person-get-phones person-1)))
```

Listing 8

```
(marry mike-sperber sabine-ferber) =>
#crud.phone_book.Person{
  :first "Mike"
  :last "Ferber"
  :phones {:home "(123) 555-1234" :work "(321) 555-4321"}}
```

Listing 9

```
(assoc {:home "(123) 555-1234" :work "(321) 555-4321"}
       :home "(123) 555-2345")
=> {:home "(123) 555-2345", :work "(321) 555-4321"}
```

Listing 10

```
(defn new-phone [person key to]
  (make-person (person-get-first person)
               (person-get-last person)
               (assoc (person-get-phones person) key to)))
```

Listing 11

Objekt her. Angenommen, eine Person bekommt eine neue Telefonnummer. Dazu muss aus der Map mit den schon vorhandenen Nummern eine neue Map gemacht werden (Verändern geht nicht), in der alle Einträge aus der alten Map sowie ein weiterer stehen. In Clojure erledigt das die Funktion „assoc". *Listing 10* zeigt ein Beispiel. Eine Funktion, die das auf „Person"-Objekten erledigt, muss die Map mit den Telefonnummern erst extrahieren, eine neue Map und damit ein neues Objekt erzeugen *(siehe Listing 11)*.

Linsen zur Daten-Manipulation

Die Funktion „new-phone" sieht etwas umständlich aus: Im Java-Programm steht einfach „person.phones.put(key, to) ". Geht das nicht einfacher? Es geht, und zwar mit einem Konzept namens „Linsen"

```
(ns phonebook
  (:require [active.clojure.record :refer :all]
            [active.clojure.lens :as lens]))
```

Listing 12

```
(define-record-type Person
  (make-person first last phones)
  person?
  [(first person-get-first person-first)
   (last person-get-last person-last)
   (phones person-get-phones person-phones)])
```

Listing 13

```
(lens/yank mike-sperber person-first) => "Mike"
(lens/shove mike-sperber person-last "Müller")
=> #phone_book.Person{
     :first "Mike"
     :last "Müller"
     :phones {:home "(123) 555-1234" :work "(321) 555-
4321"}}
```

Listing 14

```
(defn marry [person-1 person-2]
  (lens/shove person-1 person-last
              (lens/yank person-2 person-last)))
```

Listing 15

```
(defn new-phone [person key to]
  (lens/overhaul person
                 person-phones
                 (fn [old]
                   (assoc old key to))))
```

Listing 16

```
(defn new-phone [person key to]
  (lens/shove person
              (lens/>> person-phones key)
              to))
```

Listing 17

(lenses). Es erlaubt, bei zusammengesetzten Daten auf einen bestimmten Bestandteil der Daten zu fokussieren und sich nur um den zu kümmern. Eine Bibliothek für Linsen ist bei Active Clojure dabei. Die Präambel zum Code muss sie noch einbinden *(siehe Listing 12)*.

Das „:as lens" bedeutet, dass die Linsen-Funktionen unter dem Präfix „lens/" verfügbar sind. Für „Person"-Objekte sind Linsen für alle drei Felder erforderlich. Dazu wird die Record-Definition etwas aufgebohrt *(siehe Listing 13)*. Um jedes Feld herum sind also Klammern; an letzter Stelle steht jeweils der Name der Linse. Eine Linse kann sowohl dafür benutzt werden, den fokussierten Teil der Daten zu extrahieren, als auch dafür, ein neues Objekt zu erzeugen, das beim fokussierten Teil einen anderen Wert hat. Das erledigen die Linsen-Funktionen „lens/yank" und „lens/shove" *(siehe Listing 14)*. Mithilfe dieser Funktionen lässt sich „marry" kompakter als vorher schreiben *(siehe Listing 15)*.

Bei „new-phone" ist es nützlich, eine weitere Linsen-Funktion namens „lens/overhaul" zu benutzen. Sie erlaubt es, den fokussierten Teil zu ändern, also den neuen Wert abhängig vom alten zu bestimmen. Die Funktion akzeptiert eine Funktion, die den alten Wert übergeben bekommt und den neuen produziert *(siehe Listing 16)*.

Aber es geht noch besser: Der Zugriff auf einen Eintrag einer Map ist auch eine Art Fokus: Keywords fungieren auch als Linsen. Damit ist es möglich, „(assoc old key to)" durch „(lens/shove old key to)" zu ersetzen. Das allein bringt noch keine Verbesserung. Allerdings ist es möglich, Linsen zu komponieren – auf den fokussierten Teil eines Objekts kann wiederum ein Teil fokussiert werden. Die Linsen-Funktion „lens/>>" erledigt das – sie macht aus zwei (oder mehr) Linsen eine. Die Linse „(lens/>> person-phones key)" fokussiert also auf einen Eintrag innerhalb der Telefonnummern einer Person. Damit lässt sich nun „new-phone" kompakt und intuitiv schreiben *(siehe Listing 17)*. Linsen können noch eine Menge mehr; wer nach „lenses functional programming" im Internet sucht, findet es.

Fazit

Unveränderliche Daten zu benutzen, ist mehr als nur eine gute Idee: Sie eröffnen eine neue Sichtweise auf die Datenmodellierung, die auf konsistenten Schnappschüssen statt auf gekapseltem Zustand aufbaut. Sie sind eine konsequente Fortführung der ursprünglichen Idee der objektorientierten Programmierung und werden von funktionalen Sprachen besonders gut unterstützt – auch auf der Java-Plattform.

Weitere Informationen

[1] Joshua Bloch, Effective Java, 3rd edition, Addison-Wesley, 2017

[2] Alan Kay, The early history of Smalltalk, Proceedings HOPL-II The second ACM SIGPLAN conference on History of programming languages, ACM, 1993 Seiten 69 –95: *http://worrydream.com/EarlyHistoryOfSmalltalk*

[3] Active Clojure: *https://github.com/active-group/active-clojure*

Michael Sperber

michael.sperber@active-group.de

Dr. Michael Sperber ist Geschäftsführer der Active Group GmbH in Tübingen, die Individualsoftware entwickelt. Er ist seit mehr als zwanzig Jahren in Forschung über und industrieller Anwendung von funktionaler Programmierung tätig. Er hat zahlreiche Fachartikel zum Thema verfasst, Anfänger-Ausbildungen in Programmierung an den Universitäten Tübingen, Freiburg und Kiel konzipiert und (in Tübingen) mehrfach durchgeführt. Michael Sperber gehört zu den Mitinitiatoren und Autoren des Blogs „funktionale-programmierung.de" und der Entwicklerkonferenz BOB.