Developer Week 2016

EVENTSOURCING MIT F#

Carsten König

WAS IST EVENTSOURCING

nicht der **Zustand**

sondern die **Ereignisse**

die zu diesem Zustand geführt haben werden gespeichert.

BEISPIEL KONTO

ANSTATT

Speichern des aktuellen **Zustands = Guthaben**

jetzt beträgt ihr Guthaben 15€

EREIGNISSE

Werden die **Ereignisse = Ein- und Auszahlungen** gespeichert

- am `01.01` erfolgte eine `Einzahlung` von `50€`
- am `02.01` erfolgte eine `Auszahlung` von `40€`
- am `03.01` erfolgte eine `Einzahlung` von `5€`
- ...

BEGRIFFE

- **Aggregate** = das Domänen-Objekt für das wir uns interessieren (Konto)
- **Event** = beschreibt eine Änderung an einem Aggregat (Ein-/Auszahlungen)
- **Stream** = Abfolge von Ereignissen eines bestimmten Aggregats
- **Source** = Verwaltet die Streams der einzelnen Aggregate
- **Projektion** = Berechnen eines Zustands aus den Ereignissen eines Aggregats

SCHNITTSTELLEN

```
1: type IEventSource =
2:     abstract GetStream : id:AggregateId -> IEventStream
3:
4: type IEventStream =
5:     abstract Add       : event:'event -> unit
6:     abstract Enumerate : upper:VersionBound -> 'event list
```

HEUTE GEHT ES UM ...

Wie bekomme ich **Zustand** aus **Ereignissen**?

SZENARIO: SESSIONS

Sessions und Bewertungen auf einer Konferenz

MODEL:

```
 1: type Events =
 2:     | Angelegt           of Sprecher * Titel
 3:     | TitelAktuallisiert of Titel
 4:     | Gestartet          of DateTime
 5:     | Beendet            of DateTime
 6:     | Bewertet           of Teilnehmer * Sterne
 7:
 8: type Sterne =
 9:     | EinStern
10:     | ZweiSterne
11:     | DreiSterne
```

FRAGEN:

- Wie lautet der aktuelle **Titel**?
- Während welchem **Zeitraum** fand die Session statt?
- Wie sieht die **Durchschnittsbewertung** aus?

ZIEL-PROJEKTION

```
type Zusammenfassung =
    {
        Sprecher           : Sprecher
        Titel              : Titel
        AnzahlÄnderungen   : int
        Zeitraum           : Zeitraum
        Bewertung          : decimal
        AnzahlBewertungen  : int
    }
```

DIREKTER ANSATZ

rekursiv die Ereignisse durchlaufen

DEMO

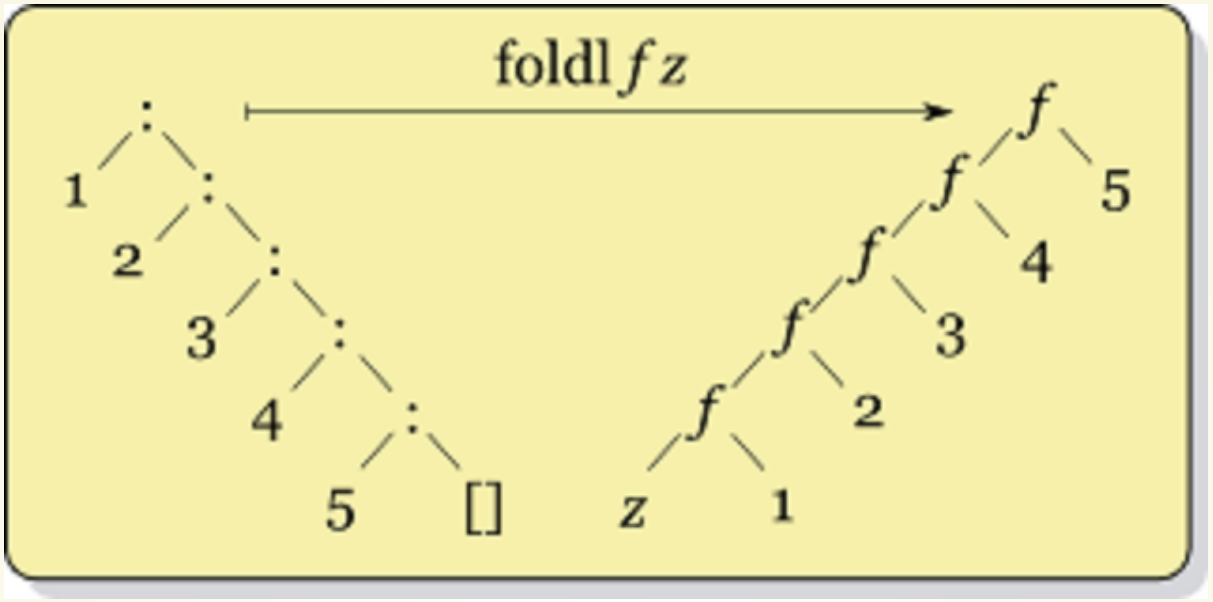
BEISPIEL

```
1: let titel (evs : Ereignisse list) : Titel =
2:     let rec letzterTitel aktTitel =
3:         function
4:         | []                              -> aktTitel
5:         | Angelegt (_,t)          :: evs -> letzterTitel t evs
6:         | TitelAktuallisiert t    :: evs -> letzterTitel t evs
7:         | _                       :: evs -> letzterTitel aktTitel evs
8:     letzterTitel (Titel "---") evs
```

NACHTEILE

- Code-Wiederholung und direkte Rekursion
- Ereignisse werden mehrfach durchlaufen
- `bewertungen` umfasst eigentlich `anzahlBewertungen`

FOLD



MUSTER ...

```
1: let rec fold f acc xs =
2:     match xs with
3:     | []      -> acc
4:     | (x::xs) -> fold f (f acc x) xs
5:
6: fold f initial ...
```

DEMO

BEISPIEL

```
 1: let bewertung : Ereignisse seq -> decimal =
 2:     Seq.fold
 3:         (fun (wert, anzahl) ->
 4:             function
 5:             | Bewertet (_,EinStern)   -> (wert+1, anzahl+1)
 6:             | Bewertet (_,ZweiSterne) -> (wert+2, anzahl+1)
 7:             | Bewertet (_,DreiSterne) -> (wert+3, anzahl+1)
 8:             | _                       -> (wert, anzahl))
 9:         (0,0)
10:     >> function
11:         | wert, anzahl when anzahl > 0 ->
12:             decimal wert / decimal anzahl
13:         | _ -> 0m
```

IMMER NOCH...

- Ereignisse werden mehrfach durchlaufen
- `bewertungen` beinhaltet eigentlich `anzahlBewertungen`

PROJEKTIONEN

MUSTER ...

```
1: Seq.fold
2:     foldFun
3:     init
4: >> proj
```

NEUE ABSTRAKTION:

```
1: type Projection<'s,'event,'result> = {
2:     Fold : 's -> 'event -> 's
3:     Proj : 's -> 'result
4:     Init : 's
```

GEÄNDERTES INTERFACE

```
1: type IEventStream =
2:     abstract Add        : event:'event -> unit
3:     abstract Read       : p:Projection<'s,'e,'r> -> upper:VersionBound -> 'r
```

KOMBINATOREN

```fsharp
let createP f i p : Projection<_,_,_> =
    { Fold = f
    ; Init = i
    ; Proj = p }

let inline sumByP (select : 'event -> 'num option) =
    createP
        (fun sum ev ->
            match select ev with
            | Some nr -> sum + nr
            | None    -> sum)
        LanguagePrimitives.GenericZero
        id

let countByP (select : 'event -> bool) =
    let toNum =
        function
        | true  -> Some 1
        | false -> None
    sumByP (select >> toNum)
```

DEMO

BEISPIEL

```
1: let anzahlÄnderungen : Projection<_,_,int> =
2:     countByP
3:         (function
4:         | TitelAktuallisiert _ -> true
5:         | _                    -> false)
```

WAS FEHLT

- Ereignisse werden mehrfach durchlaufen
- *kombinieren*

PARALLELE PROJEKTIONEN

IDEE

Übergang zu **Paaren** von *Projektionen*

- Zustand -> Zustand A * Zustand B
- komponentenweise `Fold`
- komponentenweise `Proj`

```fsharp
type Pair<'a,'b> = {
    First : 'a
    Second : 'b
    }

let parallelP
    ( pa : Projection<'sa,'event,'ra>
    , pb : Projection<'sb,'event,'rb>)
    : Projection<Pair<'sa,'sb>,'event,'ra*'rb> =
    {
        Init = { First = pa.Init; Second = pb.Init }
        Proj = fun pair ->
            (pa.Proj pair.First, pb.Proj pair.Second)
        Fold = fun pair ev ->
            let fst = pa.Fold pair.First ev
            let snd = pb.Fold pair.Second ev
            { pair with First = fst; Second = snd }
    }
```

PROBLEM

Verlieren etwas die Kontrolle über den *Ergebnis*-Typ

FUNKTOR

```
let fmap
    (f : 'a -> 'b)
    (pa : Projection<'s,'event,'a>)
    : Projection<'s,'event,'b> =
    {
        Init = pa.Init
        Fold = pa.Fold
        Proj = pa.Proj >> f
    }
```

BEISPIEL

```
 1: let zeitraum : Projection<_,_,Zeitraum> =
 2:     let von =
 3:         lastP
 4:             (function Gestartet t -> Some t | _ -> None)
 5:             DateTime.MinValue
 6:     let bis =
 7:         lastP
 8:             (function Beendet t -> Some t | _ -> None)
 9:             DateTime.MinValue
10:     parallelP (von, bis)
11:     |> fmap (fun (v,b) -> { Von = v; Bis = b })
```

DEMO

VORTEILE

- Projektionen sind kombinierbar
- Ereignisse werden nur einmal durchlaufen

NACHTEILE

```
 1: let zusammenfassung' =
 2:     parallelP (sprecher,
 3:         parallelP (titel,
 4:             parallelP (anzahlÄnderungen,
 5:                 parallelP (zeitraum,
 6:                     parallelP (bewertung, anzahlBewertungen)))))
 7:     |> fmap (fun (s, (t, (anzÄ, (z, (b, anzB))))) ->
 8:     {
 9:         Sprecher           = s
10:         Titel              = t
11:         AnzahlÄnderungen   = anzÄ
12:         Zeitraum           = z
13:         Bewertung          = b
14:         AnzahlBewertungen  = anzB
15:     })
```

APPLIKATIVER FUNKTOR

IMPLEMENTIERT

```
 1: let pureP value =
 2:     {
 3:         Init = ()
 4:         Proj = fun _ -> value
 5:         Fold = (fun _ _ -> ())
 6:     }
 7:
 8: let aMap
 9:     (pf : Projection<'sf,'event,'a -> 'b>)
10:     (pa : Projection<'sa,'event,'a>)
11:     : Projection<Pair<'sf,'sa>,'event,'b> =
12:     parallelP (pf, pa)
13:     |> fmap (fun (f,a) -> f a)
```

OPERATOREN

```
1: let (<*>) = aMap
2: let (<*) f a = (pureP f) <*> a
```

IDEE

- bringe eine Funktion `f` in *Curry*-Form mit `pureP f` in eine Projektion
- reduziere deren *Stelligkeit* mittels `aMap` der Reihe nach durch *Argument-Projektionen*

... MIT TYPEN

Erinnerung:

```
aMap : Projecton<_,_,'a->'b> -> Projection<_,_,'a>
    -> Projection<_,_,'b>
```

Wenn jetzt

```
1: f : 'a -> ('b -> 'c)
2: pa : Projection<_,_,'a>
3: pb : Projection<_,_,'b>
```

dann ist

```
1: pureP f : Projection<_,_,'a->('b->'c)>
2: pureP f <*> pa : Projection<_,_,'b->'c>
3: (pureP f <*> pa) <*> pb : Projection<_,_,'c>
```

BEISPIEL

```
 1: let zusammenfassungConst s t anzÄ z b anzB =
 2:     {
 3:         Sprecher           = s
 4:         Titel              = t
 5:         AnzahlÄnderungen   = anzÄ
 6:         Zeitraum           = z
 7:         Bewertung          = b
 8:         AnzahlBewertungen = anzB
 9:     }
10:
11: let zusammenfassung : Projection<_,_,Zusammenfassung> =
12:     zusammenfassungConst
13:     <* sprecher <*> titel
14:     <*> anzahlÄnderungen <*> zeitraum
15:     <*> bewertung <*> anzahlBewertungen
```

DEMO

SNAPSHOTS

IDEE

Der innere *Zustand* der Projektionen

```
1: type Projection<'snapshot,'event,'result> = ...
```

ist der Snapshot

```
type Snapshot<'snapshot> =
    {
        AggregateId : AggregateId
        Version     : AggregateVersion
        Value       : 'snapshot
    }
```

NEUE SCHNITTSTELLE

```
1: type IEventStream =
2:     abstract Add            : event:'event -> unit
3:     abstract TakeSnapshot   : p:Projection<'s,'e,'r> -> upper:VersionBound -> u
4:     abstract Read           : p:Projection<'s,'e,'r> -> upper:VersionBound -> '
```

PARALLELEN PROJEKTIONEN

BEISPIEL

```
1: bewertung : Projection<Pair<Pair<unit,int>int>,_,_> =
2:     ...
3:     fun wert anzahl -> ...
4:     <* summeBewertungen <*> anzahlBewertungen
```

- `unit` kann ignoriert werden
- erste `int` von `summeBewertung`
- zweite `int` von `anzahlBewertung`

PROBLEM

- wie bei unterschiedlichen **Versionen** vorgehen
- wie die beiden `int` Unterscheiden?

LÖSUNGEN VERSIONEN

im `Fold` der Projektion die Version mit übergeben

```
1: type Projection<'s,'event,'result> = {
2:     Fold : 's -> 'event * AggregateVersion -> 's
3:     Proj : 's -> 'result
4:     Init : 's
5:     }
```

`Pair<..>` erweitern:

```
1: type Pair<'a,'b> = {
2:     First  : AggregateVersion * 'a
3:     Second : AggregateVersion * 'b
4:     }
```

und in `parallelP` nur bei höherer Version *folden*:

```
 1: let parallelP ... =
 2:     {
 3:         Init = ...
 4:         Proj = ...
 5:         Fold = fun pair (ev, ver) ->
 6:             match pair with
 7:             | { First = (verA,sA); Second = (verB, sB) } ->
 8:                 let fst =
 9:                     if ver > verA
10:                     then (ver, pa.Fold sA (ev, ver))
11:                     else (verA, sA)
12:                 let snd = ...
13:                 in { First = fst; Second = snd }
14:     }
```

LÖSUNG DOPPEL-`INT`

Phantomtypen einführen

```
 1: type Summe<'label,'a> = Summe of 'a
 2:
 3: let inline sumByP
 4:     (label : 'label)
 5:     (select : 'event -> 'num option)
 6:     : Projection<Summe<'label,'num>,_,'num> =
 7:     createP
 8:         (fun (Summe sum) (ev,_) ->
 9:             match select ev with
10:             | Some nr -> Summe (sum + nr)
11:             | None    -> Summe sum)
12:         (Summe LanguagePrimitives.GenericZero)
13:         (fun (Summe num) -> num)
```

und zur Unterschiedung verwenden

```
1: type AnzahlBewertungen = AnzahlBewertungen
2: let anzahlBewertungen : Projection<_,_,int> =
3:     countByP AnzahlBewertungen
4:         (function
5:             | Bewertet _ -> true
6:             | _          -> false)
```

damit hat `bewertung` den Typ

```
1: Projection<
2:     Pair<
3:         Pair<
4:             unit,
5:             Summe<SummeBewertungen,int>
6:         >,
7:         Summe<AnzahlBewertungen,int>
8:     >
9: >
```

DEMO

VIELEN DANK!

Carsten König

- Twitter: CarstenK_dev
- Email: Carsten@gettingsharper.de
- Web: gettingsharper.de