Event Sourcing
- Versionskontrolle für Ihre Daten -

Philip Jander
Jander.IT

Quellcode: github.com/janderit/dwx2015eventsourcing

Event sourcing

> statt des Systemzustands:
> die Historie aller Änderungen persistieren,
> die zum Zustand geführt haben.

Welche SQL Anweisung(en) löschen Daten?

[ ] INSERT
[X] UPDATE
[X] DELETE
[ ] SELECT

```
> update ... set A=456;
```

SQL Tabelle:

```
  A
|---|
 456   <- was stand hier vor dem UPDATE?
```

Daten(bank)schema

Rechnungs-Id	Rechnungsdatum	Belegnummer	Netto	USt	Offen
4711	08.03.2015	R-715.32	132.86	19%	132.86
4712	08.03.2015	R-719.30	17.45	19%	17.45
4713	09.03.2015	R-799.12	86.50	19%	0.50

```
{
    "Rechnungs-Id": 4711,
    "Rechnungs-Datum": "08.03.2015",
    "Belegnummer": "R-715.32",
    "Netto-Betrag": 132.86,
    "USt": 19%,
    "Offener Betrag": 132.86
}
```

Aber was ist, wenn sich das Schema ändert?

-> Datenmigration :-(

```
Datenschema:     Eingeschränkte Informationen erfassen
Mutable state:   Daten löschen (um Speicherplatz zu sparen?!)

Event sourcing:  zunächst maximal verfügbare Daten erfassen,
                 bei Bedarf interpretieren und temporär reduzieren
```

Warum ist Löschen von Daten problematisch?

+ Änderungen (vollständig & garantiert) nachvollziehen
+ Korrektur von Fehlern
+ Neue Anforderungen

Alternative:
Ereignisse aufzeichnen (wer war beim 'Event Storming' Vortrag?...)

Geschäftsprozessanalyse liefert Ereignisse...
Ereigniskette 'Bedienung Debitorenrechnung'

Debitorenrechnung ist eingegangen
-> Zahlungsausgang wurde Debitorenrechnung zugeordnet
-> Rechnung wurde vollständig beglichen
Vorsteuerabzug wurde begründet

Debitorenrechnung ist eingegangen
Zeitpunkt: 2015-03-09T13:17:01
Bearbeiter: 43
Id: 4711
Lieferant: 815
Rechnungsdatum: 2015-03-08
Beleg: R-713.92
Fremdbeleg: R220301099/4
Nettobetrag: 132.41
Vorsteuer: 26.00

Zahlungsausgang wurde Debitorenrechnung zugeordnet
Zeitpunkt: 2015-03-10T08:35:22
Bearbeiter: 43
Zahlung: 103872
Rechnung: 4711
Buchungstag: 2015-03-11
Betrag: 158.41

Rechnung wurde vollständig beglichen
Rechnung: 4711
Buchungstag: 2015-03-11

Vorsteuerabzug wurde begründet
Datum: 2015-03-11
Betrag: 26.00
Rechnung: 4711
Beleg: R-713.92

Aktueller Zustand: Projektion

```
                                                bei Bedarf
[Ereignis],[Ereignis],[Ereignis],[Ereignis]       ( => )    Zustand
```

Offener Betrag Rechnung 4711 :=

```
    Debitorenrechnung ist eingegangen.Nettobetrag               132.41
  + Debitorenrechnung ist eingegangen.Vorsteuer                  26.00
  - Zahlungsausgang wurde Debitorenrechnung zugeordnet.Betrag   158.41

  =                                                               0.00
```

Erster Event Sourcing Einsatz

Luca Pacioli: Doppelte Buchführung

1494, Italien

Hauptbuch der Doppelten Buchführung

Nr.	Datum	Betrag	Beleg	SollKto	HabenKto	Bemerkungen
81601	1494-08-03	3 Gld 11	94/1982 L	Verbdl.	Aufw.	-/-
81602	1494-08-03	1 Gld --	94/8271 K	Kasse	Handk.	-/-

neue Buchung -> neuer Eintrag (Zeile) im Hauptbuch

Ereignisse sind, einmal verzeichnet, unveränderlich

das Dokument eines in der Vergangenheit liegenden Faktums

Ihre Interpretation ist wandelbar, nicht jedoch das Faktum an sich

Event sourcing: Die Liste aller verzeichneten Ereignisse
ist die *einzige* Quelle für den Systemzustand

Datenfluss mit Ereignissen

Beispiel Geschäftslogik (Pseudocode):

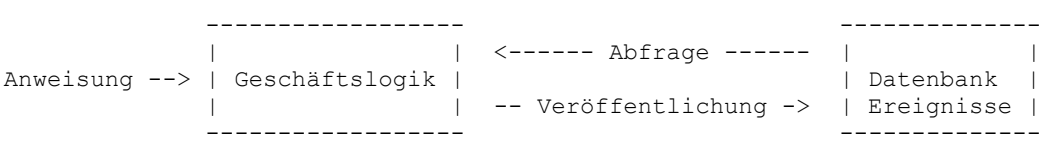
```
void Zahlung_für_Rechnung_anweisen(
  Rechnung rechnung,
  Betrag betrag,
  Konto konto)
{

    if (Offener_Betrag(rechnung)-betrag<=0)
        Fail("Kreditorenrechnungen dürfen nicht übreerfüllt werden.");

    var zahlungs_id = Zahlung.Neu;

    Ueberweisung_beauftragen(
        zahlungs_id,
        konto,
        Zahlungsempfaenger(rechnung),
        betrag);

    Zahlungsausgang_wurde_Debitorenrechnung_zugeordnet(
        zahlungs_id,
        rechnung);
}
```



Projektion:

```
Debitorenrechnung_ist_eingegangen,
Zahlung_Debitorenrechnung_zugeordnet &
Rechnung_wurde_vollständig_beglichen

->

Offener Betrag einer Rechnung

1. naiv:

var offenerbetrag = 0;
foreach (var e in historie){
    if (e is Debitorenrechnung_ist_eingegangen) offenerbetrag+=e.Betrag;
    if (e is Zahlung_Debitorenrechnung_zugeordnet) offenerbetrag-=e.Betrag;
    if (e is Rechnung_wurde_vollständig_beglichen) offenerbetrag=0;
}


2. Besser:


var offenerBetrag = Projektion.Neu(startwert: 0)
          .Fuer<Debitorenrechnung_ist_eingegangen>
             ((x,e)=>x+e.Betrag)
          .Fuer<Zahlung_Debitorenrechnung_zugeordnet>
             ((x,e)=>x-e.Betrag);

var betrag = historie.Aggregate(offenerBetrag.Startwert,offenerBetrag.Iteration);
```

Event sourcing: Die Liste aller verzeichneten Ereignisse
ist die *einzige* Quelle für den Systemzustand

-> Schema leitet sich idealierweise direkt aus der
   Problemdomäne ab

-> keine Korruption des Modells durch Implentierungsansätze

-> Ereignisse und ihre Beschreibung sind i.d.Regel extrem stabil gg. Änderungen

-> Die Ereignisse enthalten *garantiert* alle für Nachvollziehbarkeit
   des Systemgeschehens relevanten Vorinformationen

-> Die Ereignisse sind ein vollständiges Protokoll aller Vorgänge

-> Nachträgliche Auswertungen möglich

Nachträgliche Auswertungen:

[Produkt A in Warenkorb]
[Produkt B in Warenkorb]
[Produkt B aus Warenkorb]

Warenkorb: A

[Check-out]

Warenkorb: -

ABER:

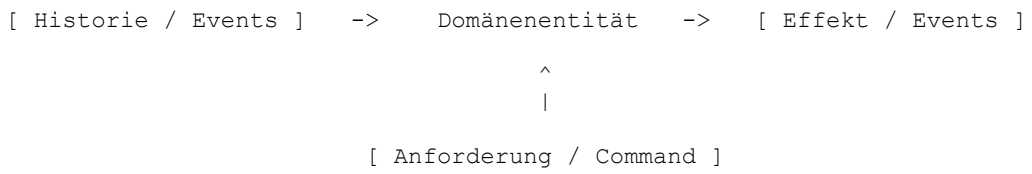
Produkte, die der Kunde kurz vor dem letzten
Checkout wieder entfernt hat: (B)

Kollaborative Domänen: Konflikte sind zur Entwurfszeit lösbar

[Kunde hat Rechnung beglichen] <-vs-> [Mahnung wurde erstellt]

Event Sourcing ist kompatibel zu DDD und CQRS

DDD (domain driven design)

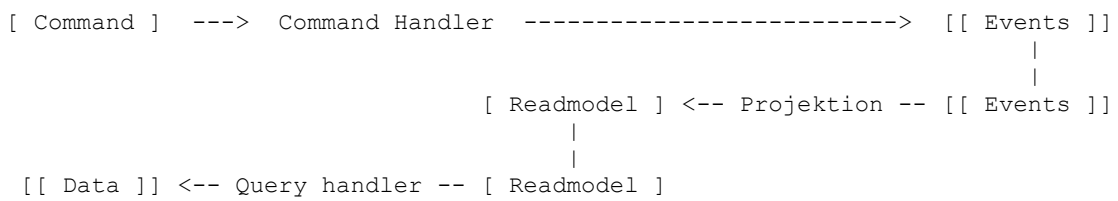


Testen mit Ereignissen:

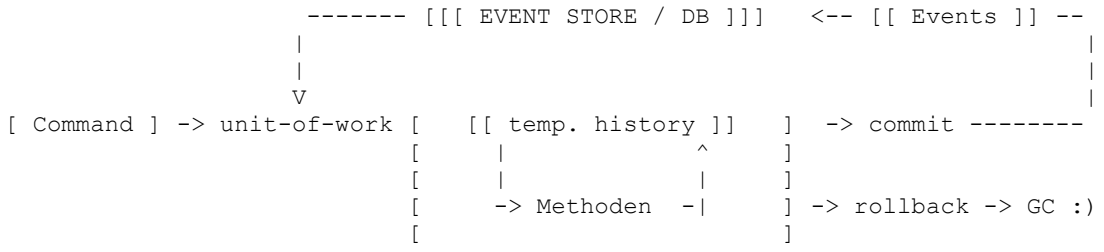
Gegeben sei die Historie:
    'Ein Benutzerzugang xyz wurde erstellt',
    'Der Benutzerzugang xyz wurde gesperrt'.
Wenn die Aktion 'Benutzer xyz meldet sich am System an' ausgeführt wird,
dann erwarte das Ereignis
    'Benutzeranmeldung für xyz ist fehlgeschlagen'.

(Abwesenheit nicht spezifizierter Effekte ist nachweisbar)

CQRS (command/query responsibility segregation):



Die Wirklichkeit ist etwas komplizierter...



Performance

1. on-demand: daten = Projektion (alle_events)

2. on-demand, partitioniert:
daten = Projektion (events_fuer(id))

3. voraggregiert, in-memory:
```
snapshot = Projektion ( events_fuer(id) )
on_event ( e=> snapshot.handle(e) )
data = snapshot.current
```

4. voraggregiert, on-disk:
```
save_to_db (Projektion ( events_fuer(id) )
on_event ( e=> save_to_db(load_from_db.handle(e)) )
data = load_from_db.current
```

Größtes Performance Problem: Deserialisierung

Physische Begrenzung: Arbeitsspeicher

Lösung: entweder Snapshots, oder Sharding

Hilfreich: Events an sich sind unveränderlich,
die Historie als Liste ist append-only

Embracing change: Versionierung

Bestehende Daten bleiben gültig

Neue Ereignisse kommen hinzu,
bestehende Ereignisse werden nicht mehr neu erzeugt

Datenfehler -> Korrekturevents

Datenschutz und Löschung

Ansätze

- Markieren von Ereignissen als 'gesperrt'
- Anonymisierung / Pseudonymisierung
- Löschen von Ereignisketten

Mehr zu Event Sourcing

[1] Martin Fowler's pattern repository
http://martinfowler.com/eaaDev/EventSourcing.html

[2] Persistenz mit Event Sourcing (heise developer)

http://www.heise.de/developer/artikel
/Persistenz-mit-Event-Sourcing-1974051.html

>>> http://bit.ly/event_sourcing <<<

Quellcode: github.com/janderit/dwx2015eventsourcing

Vielen Dank ! Fragen ?

Philip Jander www.Jander.IT @ph_j