2. DOMAIN-DRIVEN-DESIGN

Versucht die Fachlichkeit als Treiber für die Technologie aufzuarbeiten.

> DDD ist Design, nicht Implementierung

Modellieren & Strukturieren der Software, indem wir die Fachlichkeit in den Vordergrund stellen.

- ✗ Eric J. Evans, 2003, „DDD" (The Blue Book)
  - ↳ Java war 8 Jahre
  - ↳ C# grad rausgekommen
  - } Objektorientierung

Objektorientierung ist toll, weil: Objekte, die haben Eigenschaften und Methoden. z.B.

robot.walk()

- ✗ Vaughn Vernon, 2013, „DDD" (The Red Book)
- ✓ Vlad Khononov, 2021, „DDD"

BUILDING BLOCKS OF DOMAIN (BUSINESS) EVENTS

- Events müssen in Vergangenheit formuliert sein, weil wir eine Story erzählen wollen.
  Ist der Kugelschreiber erstmal runtergefallen, kann zwar der Effekt durch Aufheben kompensiert werden, aber das wäre dann ein neues Event!
- Stories leben von den Verben

  a hat b gefressen — unwichtig / wichtig

  Das spannende sind die Verben, Verben gehen aber in der Fachlichkeit über

  CREATE
  READ
  UPDATE
  DELETE
  hinaus

  Es war einmal …

  Die Person hat ein neues To-Do auf die Liste gesetzt, dann hat sie festgestellt, dass sie das To-Do anpassen musste, …

  → Das wäre mit create & update nicht so spannend ☺
- Events in Passiv & Aktiv ok, je nach Schwerpunktsetzung:

  Michael stellte eine Frage
  vs.
  Eine Frage wurde von Michael gestellt

Granularität

Titel ändern
Beschreibung ändern
vs. To-Do ändern

nur 2 Events, wenn Fachlichkeitsprozess das erfordert, zB weil user E-Mail Notification bei Titeländerung bekommen soll, nicht aber bei Beschreibung.

↳ Also nicht in Datenbankfeldern denken, sondern in Aktionen, Prozessen, Events

* Ist nicht in Stein gemeißelt, Events können fortlaufend in Granularität angepasst werden.

Commands → Events

- Kann auch 1:n sein, zB

  start group (command) → group started + group joine

- Und selbes Event kann von verschiedenen Commands getriggert werden

  join group → group joined
- Verben- & Nomenstellung
  Commands = Verb vorne
  Events = Nomen vorne
  } The Native Web
- EVA Prinzip
  - ↳ Commands sind die Eingabe
  - ↳ Verarbeitung – Command-Handlin
  - ↳ Events sind die Ausgabe

[kurzlebig] Command → [State: dauerhaft] Command-Handler → Event(s)

Command-Handler: kann Funktion, Klasse mit execute Methode … sein

↪ Daten

State: zB Liste aller bereits notierten To-Dos

→ So kann der Handler entscheiden, ob es zB schon ein To-Do mit Titel x gibt.

trigger update State

Command → Command-Handler → Event(s)

func: (Command, State) → [ ] Event

Event: To-Do hinzugefügt

Validieren + update State → Liste To-Do geupdated

- AGGREGATES
  - ↳ sind eine Konsistenzgrenze, Bereiche
    Wir wollen den State auf eine gewisse Größe bringen, um ihn konsistent halten zu können.
  - ↳ Aggregates garantieren die Konsistenz ihres States, in dem sie Commands sequenzialisieren.
    Aggregate-übergreifend gilt das nicht.
  - ↳ Ein Aggregate (d.h. der darin enthaltene State) soll so klein wie möglich und so groß wie nötig sein

    so klein wie möglich: wegen dem Parallelisieren der Commands

    so groß wie nötig: wegen der Wahrung der Konsistenz