Freie Universität Berlin

# Refaktorisierung eines Produktes durch Anwendung von Domain-Driven Design

## Verteidigung der Bachelorarbeit

# Übersicht

- Das Produkt: BETA
- Motivation
- Domain-Driven Design
- Durchführung der Refaktorisierung
- Zustand nach der Refaktorisierung
- Fazit

# BETA - Kernfunktion

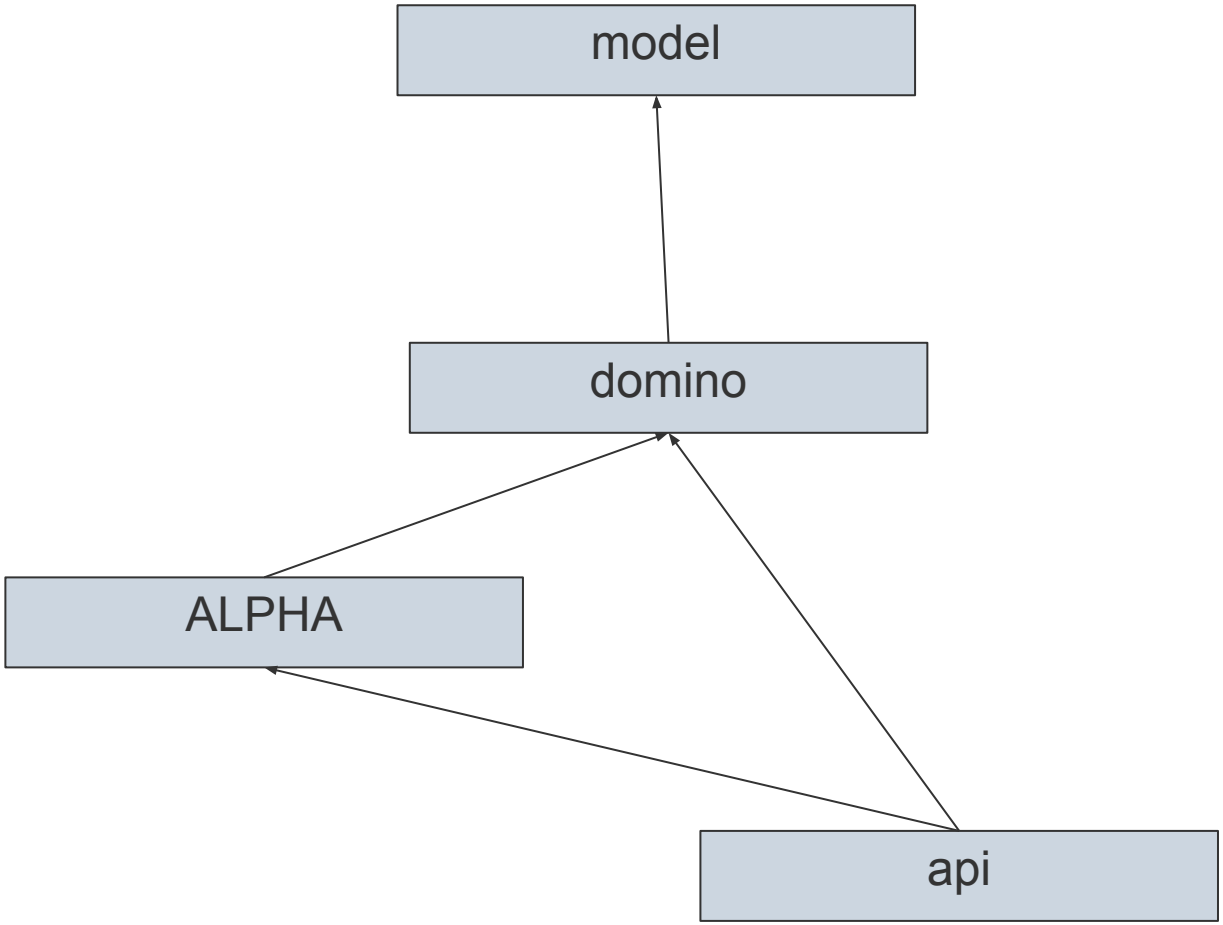
- Übertragung eines Dokumentenstapels
- Texterkennung über Optische Zeichenerkennung (mit Tesseract von Google)
- Segmentierung des Dokumentenstapels in Teildokumente
- Klassifizierung der Teildokumente
- Korrektur der Segmentierung und Klassifizierung in Weboberfläche

# Motivation

- Das Produkt BETA wird immer komplexer
- Aktuelles Modell lässt sich schwer in ihre eigentlichen Kontexte trennen
- Besonderheiten verschiedener Kunden-Prozesse haben sich vermischt
- Neuentwicklungen sind dadurch schwerer

- Ziel: Restrukturierung des Modells, sodass neue und bestehende Module besser entwickelt und gewartet werden können

# Ist-Architektur (Module)



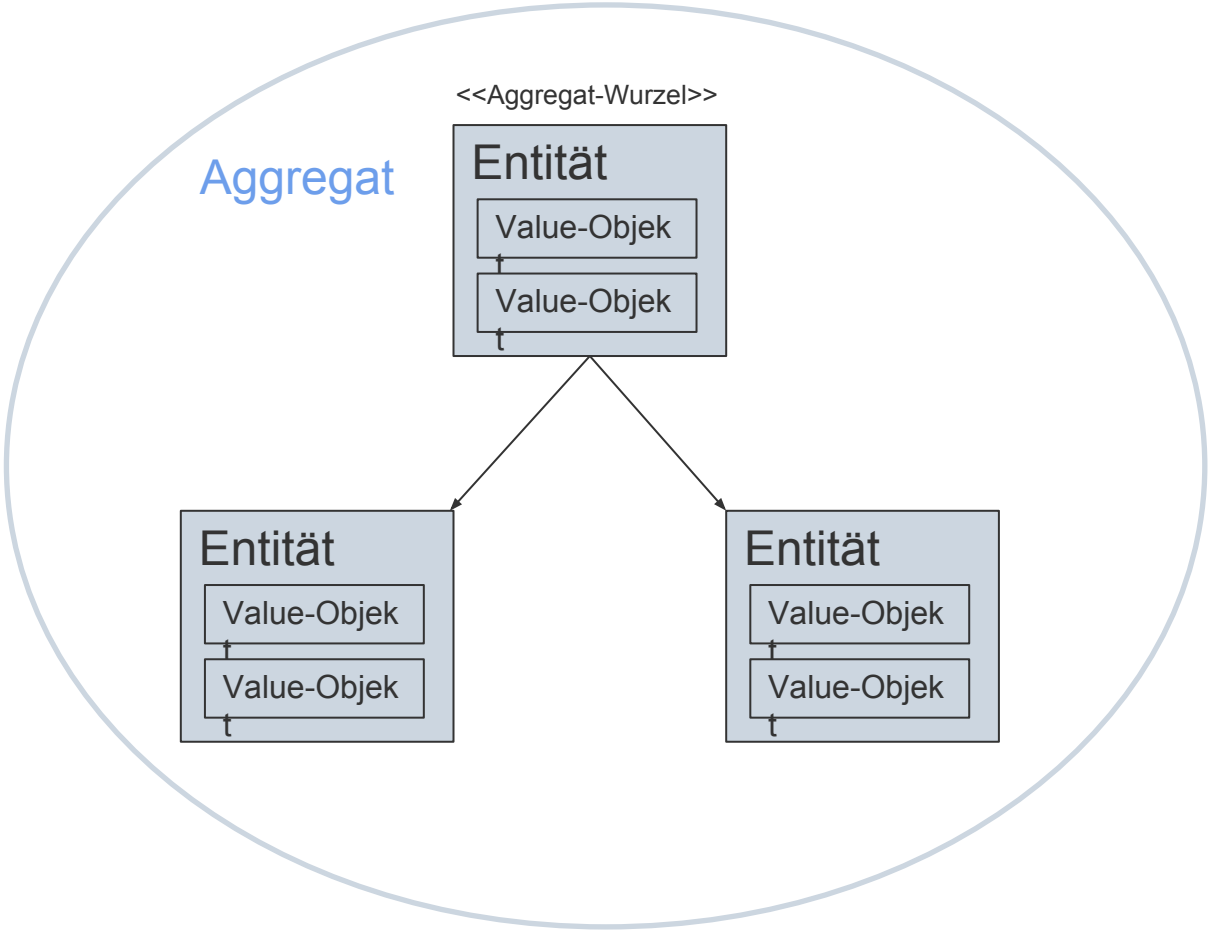
# Domain-Driven Design

- Herangehensweise für die Erstellung komplexer Software durch
  - die **Ubiquitäre Sprache**
    - Sprache der Domäne, wird in **allen** Bereichen gesprochen (Bei Gesprächen, im Code, in Dokumenten, …)
  - die Ermittlung von **Bounded Contexts**
    - Kontextuelle Grenzen innerhalb eines Prozesses
    - Beispiel: Onlineshop: Bestellung, Bezahlung, Lieferung

# Domain-Driven Design

- Herangehensweise für die Erstellung komplexer Software durch
  - Durch die Ermittlung von **Context-Maps**
    - Beschreibung der Beziehung zwischen den Bounded Contexts
      - Beispiele: Shared Kernel, Domain-Events
  - Durch Werkzeuge, wie Bounded Contexts aufgebaut werden können

# Domain-Driven Design



# Domain-Driven Design

- **Entitäten**
  - Bildet eine fachliche Einheit ab
  - Besitzt eine Identität
  - Können andere Entitäten referenzieren

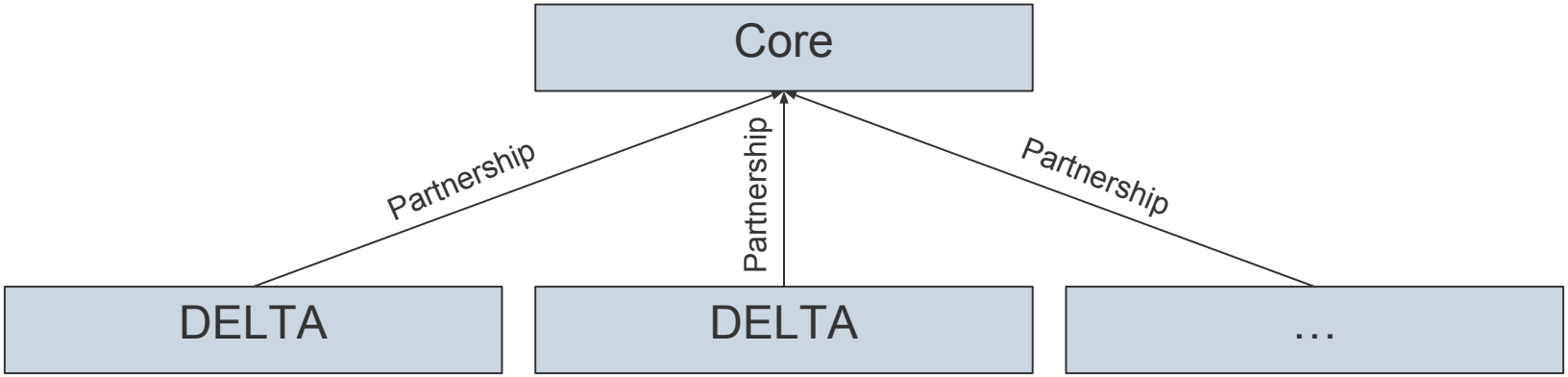
- **Value-Objekte**
  - Bilden Fachwerte ab
  - Sind unveränderlich
  - Haben keine Identität
  - Bestandteil der Entitäten

# Domain-Driven Design

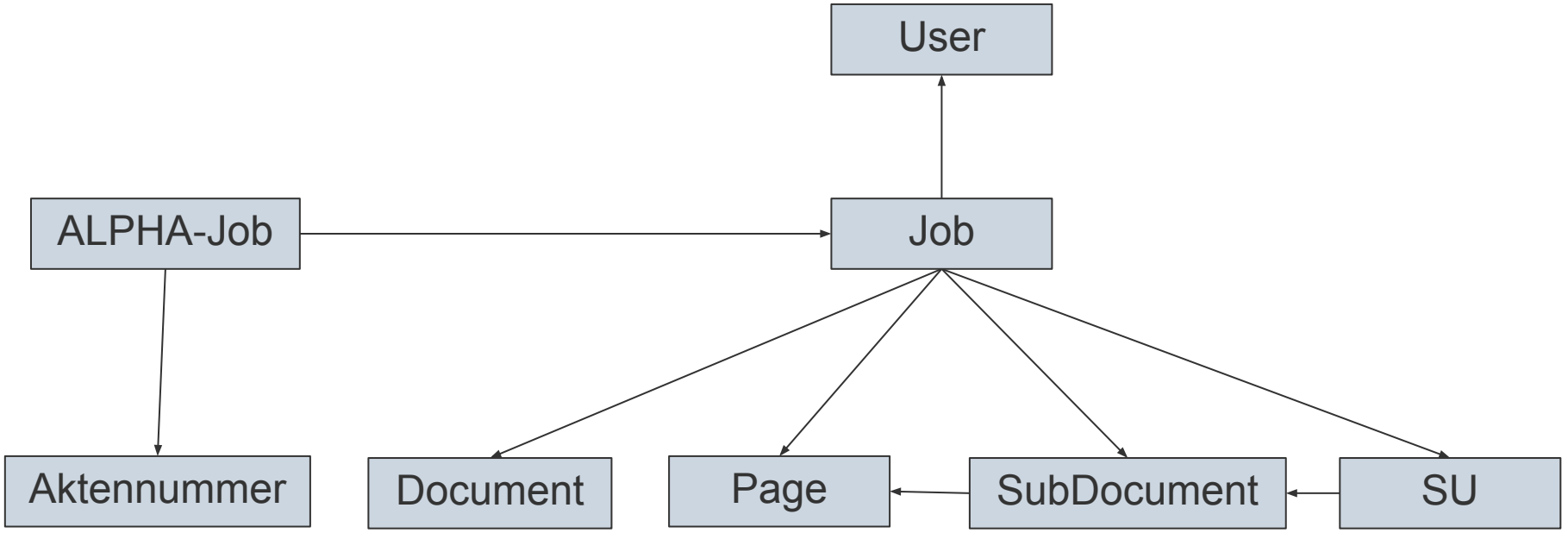
- **Aggregate**
  - Bildet ein fachliches Konzept ab
  - Besteht aus einem oder mehrerer Entitäten
  - Hat eine Wurzel-Entität
    - Hat meist den gleichen Namen wie das Aggregat
  - Ist global identifizierbar

- **Repositories**
  - Speicherort für Aggregate
  - Meist als Interface definiert
    - Technische Implementierung durch Applikation (bei BETA: MySQL + JPA)

# Sollarchitektur (Module, Bounded-Contexts)



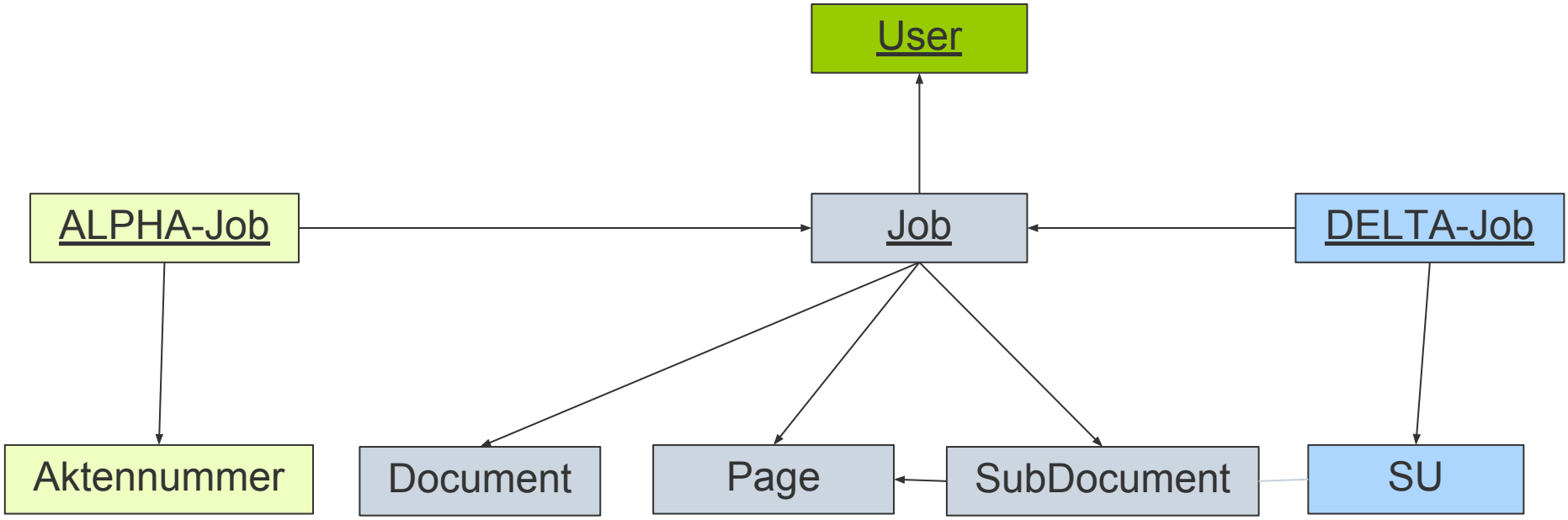
# Entitäten bei BETA



# Definition von Aggregaten

- Definition von X-Jobs: Ausprägung der BETA-Jobs für die entsprechenden Kunden
- Auslagerung der SU-Entität zu dem DELTA-Job
- Die X-Jobs sollen die Aggregat-Wurzeln sein

# Definition von Aggregaten




Gleiche Farbe: gleiches Aggregat
Unterstrichen: Aggregat-Wurzel

# Definition von Aggregaten

- Laut Domain-Driven Design soll es nur für Aggregat-Wurzeln Repositories geben
- Alle Kind-Repositories werden entfernt
- Das Managen der Kind-Entitäten muss nun die Aggregat-Wurzel übernehmen
- Die Eltern-Entitäten sollen dadurch technische und fachliche Validierung die Korrektheit des Aggregates gewährleisten
- Entitäten werden dadurch komplexer

# Definition von Aggregaten

```
public class Job {
    // ...
    @JoinColumn (name = "job_id")
    @OneToMany (cascade = CascadeType.ALL, orphranRemoval = true)
    private List<SubDocument> subDocuments;

    // laden von Entitäten
    public SubDocument getSubDocument(int subDocumentId) {
        return subDocuments.stream()
                           .filter(s -> s.getId() == subDocumentId)
                           .findAny()
                           .orElseThrow(() -> new NoSuchElementException());
    }

    // hinzufügen von Entitäten
    public void addSubDocument(SubDocument subDocument) {
        Validate.isNotNull(subDocument);
        Validate.isFalse(subDocuments.contains(subDocument));
        // fachliche Validierung, falls nötig
        subDocuments.add(subDocument)
    }

    // löschen von Entitäten...
}
```

# Übertragung von Logik zu den Aggregaten

- Logik der Entitäten waren in verschiedenen Services verstreut
- Manche Services waren überladen mit verschiedenen Aufgaben

- Übertragung der Fachlogik hin zu den Entitäten

# Übertragung von Logik zu den Aggregaten

- weniger und kleinere Services
- Logik dort, wo die Daten sind
- Beispiel: „Ein Dokument darf nur dann einem Job hinzugefügt werden, wenn er leer oder bereit ist“

```
public class Job {
    // kann EMPTY, REDY, IN_PROGRESS, FINISHED oder ERROR sein
    private Status status;

    public void addDocument(Document document) {
        Validate.isNotNull(document);

        if (status != Status.EMPTY || status != Status.READY)
            throw new IllegalStateException(„Fehlermeldung…“);

        // finde fachliche identität

        documents.add(document);
    }
}
```

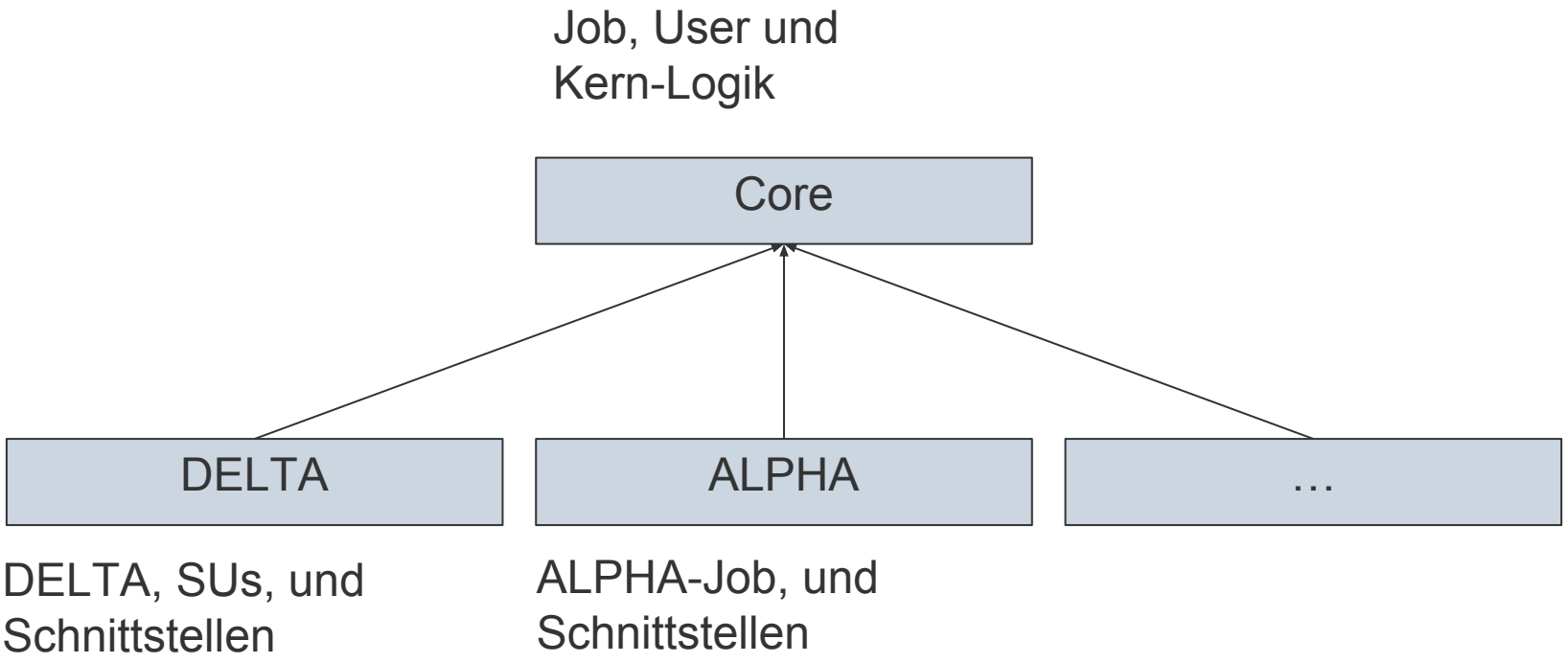
# Weitere Refaktorisierungsschritte

- Entfernung von Rückreferenzen
- Implementierung fachlicher Identitäten
- Definition einer Einheitlichen Packetstruktur

# Zustand nach der Refaktorisierung

- Lose Kopplung zwischen den Aggregaten
- Tests rund um die Aggregate sind deutlich einfacher geworden
- Entitäten haben nun mehr als nur eine Daten-Speicher-Funktion: Sie bilden nun einen Großteil der Logik ab
- Aggregate, Repositories, etc. können in die entsprechenden Soll-Module verschoben werden

# Sollarchitektur (Module)



# Fazit

- Domain-Driven Design eignet sich gut für Produkte, die einen komplexen Prozess abbilden
- BETA bildet für verschiedene Kunden deren Prozesse ab, Domain-Driven Design ist ein gutes Werkzeug dafür diese strukturiert abzubilden
- Neue Prozesse können unabhängig von den andern Kunden-Prozessen entwickelt werden

- Domain-Driven Design ist für kleinere Produkte vermutlich übertrieben
- Ist für Produkte mit dem Fokus auf einen komplexen Datenhaushalt vermutlich ungeeignet (z.B. DELTA)

Danke für ihre Aufmerksamkeit!

# WELCHE FRAGEN HABEN SIE NOCH?