Mit Domain-Driven Design (DDD) nützliche und flexible Software bauen

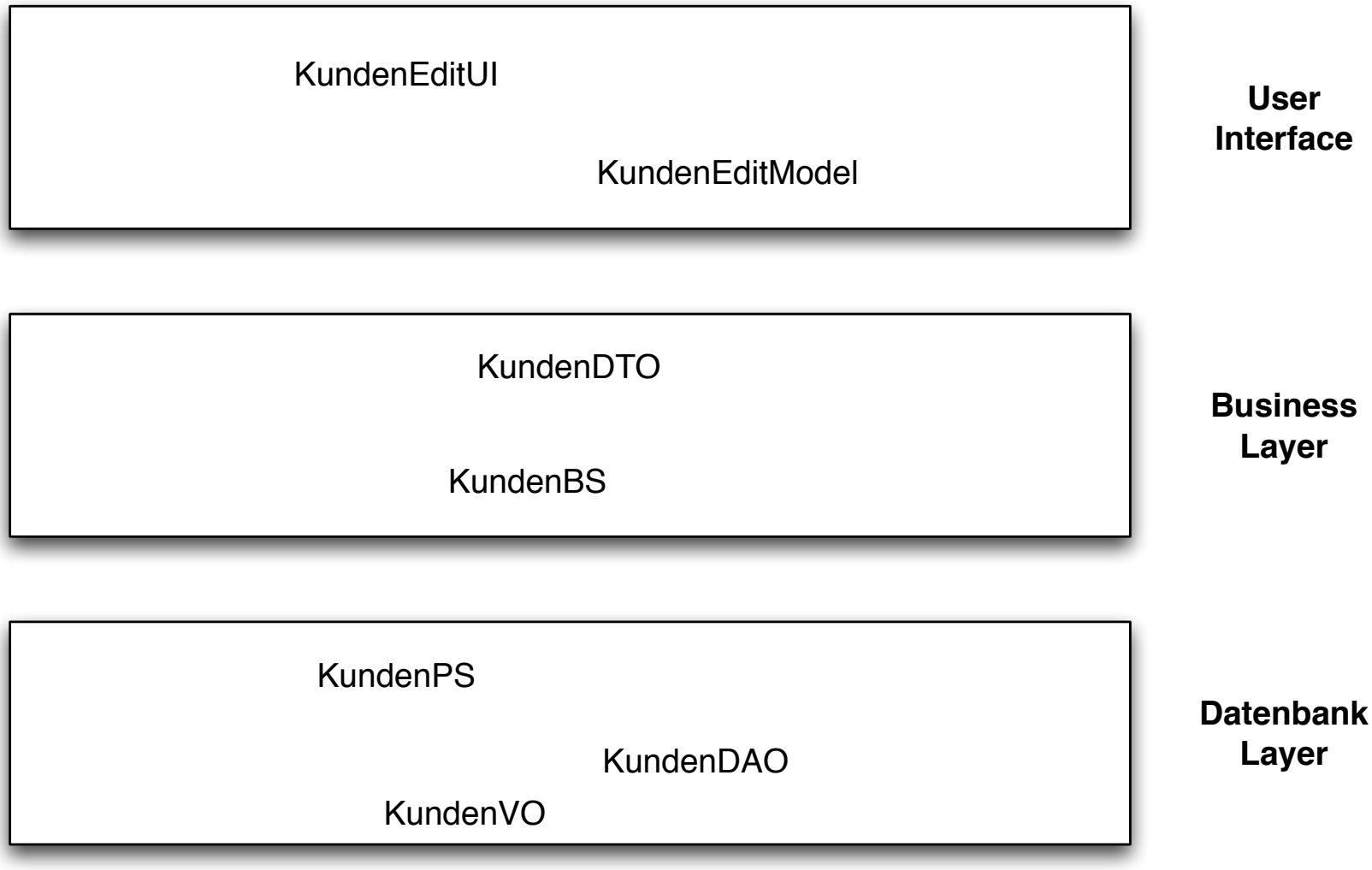
Konzepte und Methodik im Überblick

Nicole Rauch und Arif Chughtai

2. März 2016

Unsere aktuelle Architektur

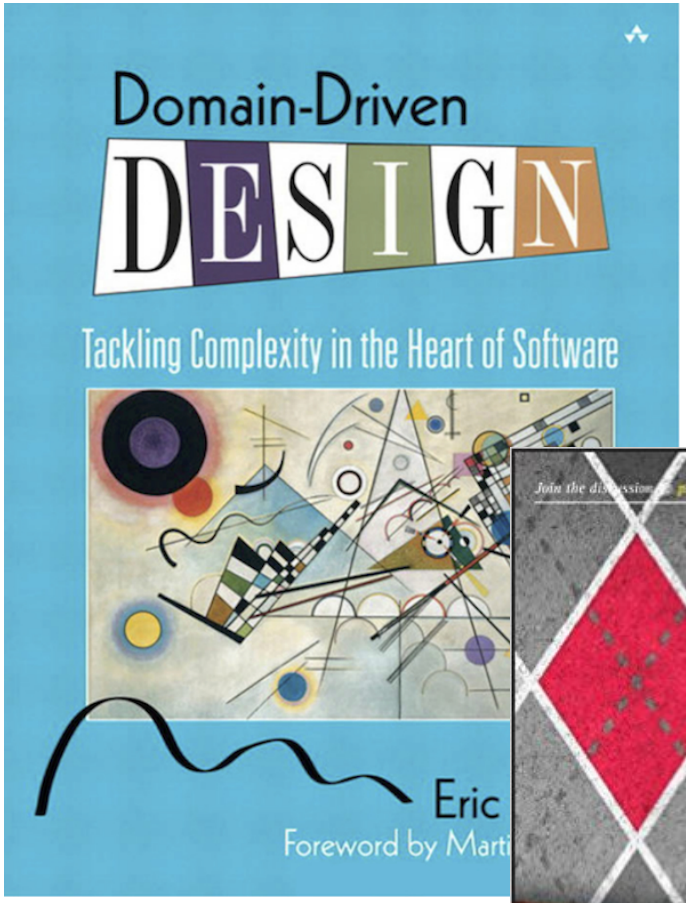
Unsere aktuelle Architektur

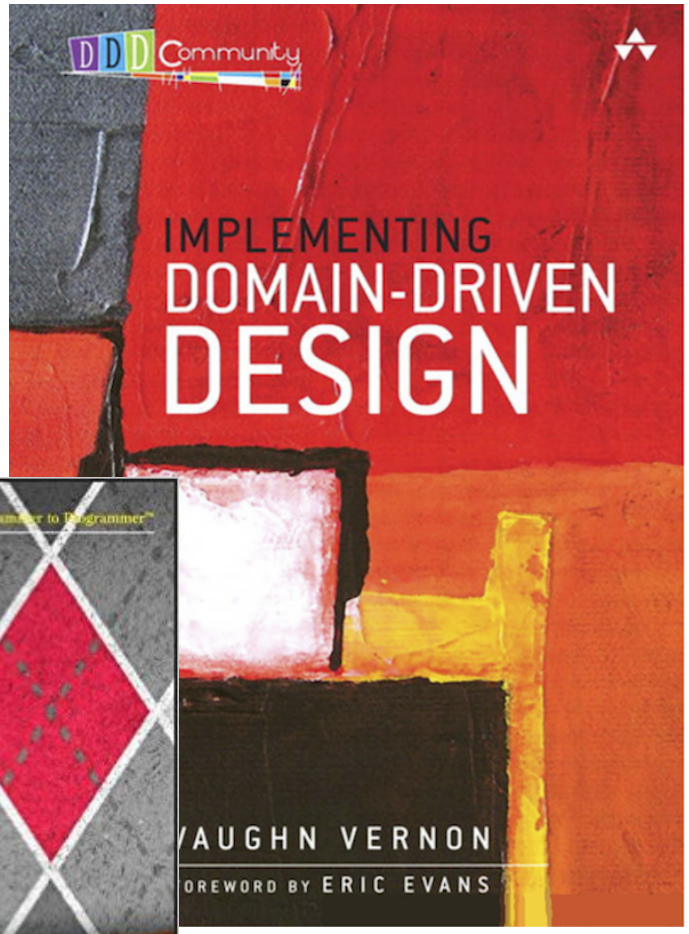


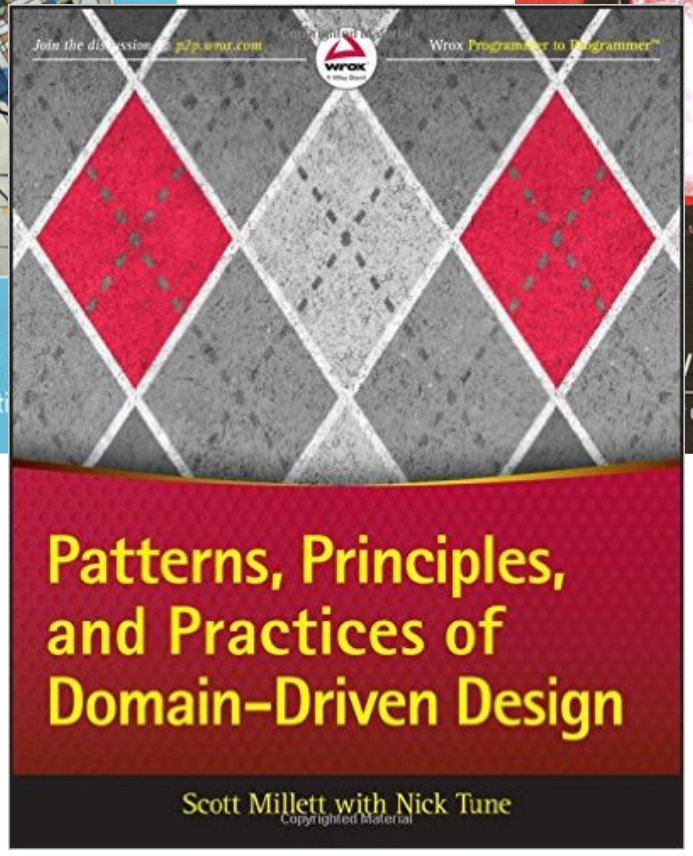
Domain-Driven Design

- Domänen-Experten und Entwickler gemeinsam
- Fokus der Entwicklung auf die Fachlichkeit
- Bausteine und Werkzeuge für gute Anwendungen

Literatur


Domain-Driven
DESIGN
Tackling Complexity in the Heart of Software
Eric
Foreword by Marti


DDD Community
IMPLEMENTING
DOMAIN-DRIVEN
DESIGN
VAUGHN VERNON
OREWORD BY ERIC EVANS


Wrox Programmer to Programmer
Patterns, Principles,
and Practices of
Domain-Driven Design
Scott Millett with Nick Tune
Copyrighted Material

Baustein: Entität

- hat eine Identität (beschreibt das „wer“)
- hat einen Lebenszyklus
- Modellierung fokussiert darauf
- eindeutigen Identifikator festlegen

Baustein: Value Object

- Wert
- hat keine Identität (beschreibt „was“, nicht „wer“)
- fachlicher Wrapper um technische Datentypen
- bildet eine konzeptionelle Einheit
- kann oft als Immutable implementiert werden
  - dann ist Sharing möglich

Einige Wochen später...

Neues über Value Objects

- aus Entitäten auslagern
- können auch komplexer aufgebaut sein
- entwickeln sich zu „Code-Magneten“

Die allgegenwärtige und gemeinsame Sprache

- Ubiquitous Language
- Fachbegriffe überall verwenden, auch im Code!
- Glossar zur Begriffsklärung
- muss reichhaltig genug sein für sämtliche Kommunikation
- beschreibt nicht nur die Einheiten im System, sondern auch Aufgaben und Funktionalitäten
- muss widerspruchsfrei und eindeutig sein
- Bounded Context, um grosse Domänen zu strukturieren

Baustein: Aggregate

- besteht aus Value Objects und Entities
- Aggregate Root als Einstiegspunkt
- gemeinsame Invarianten und Konsistenzbedingungen
- Zugriff auf Elemente über die Root

Domain Services

- Hinweis auf fehlenden Service: Code in static Methoden
- zustandslos
- oft als Einstieg in die Domäne

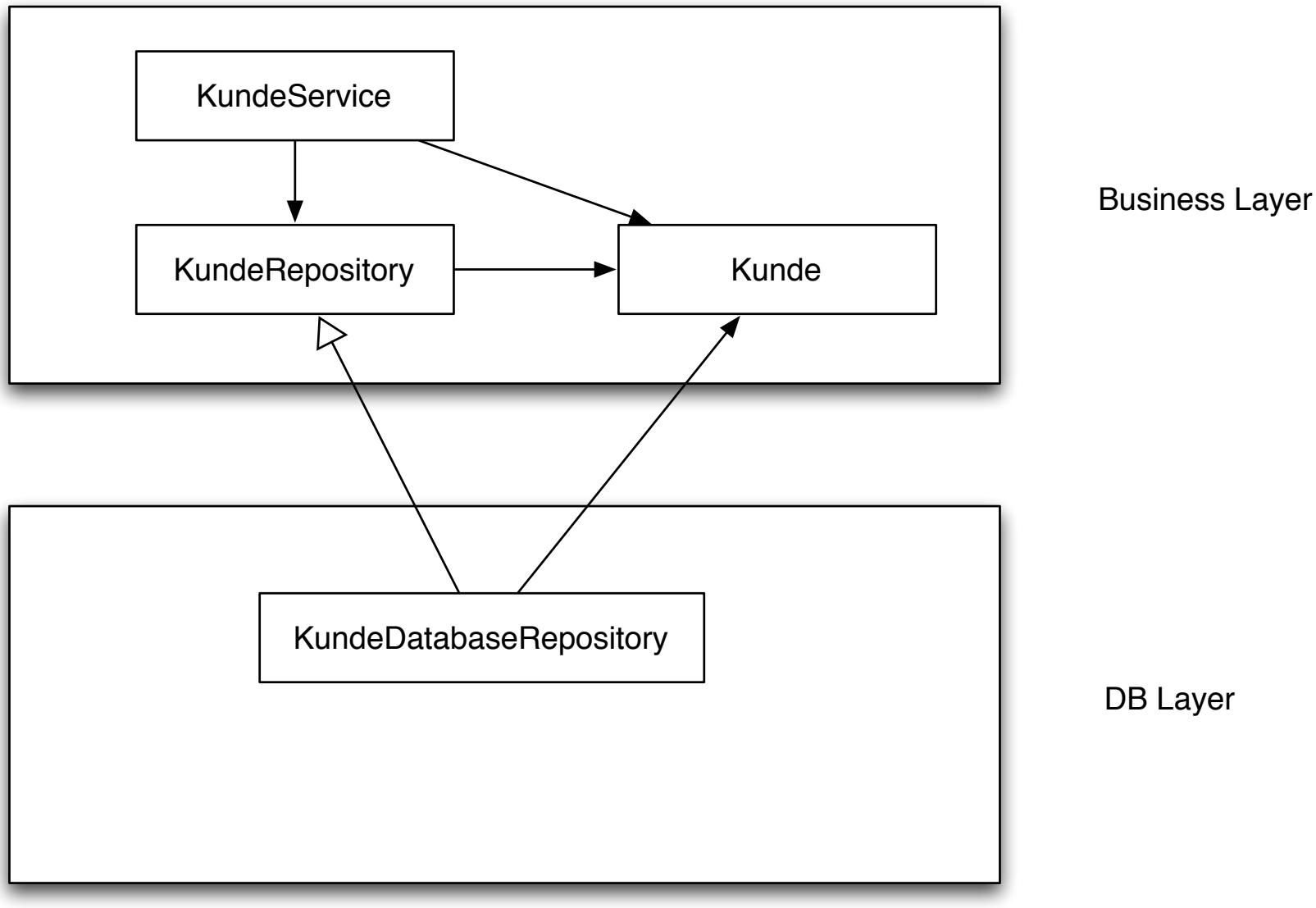
Domain Services Revisited

- zu viel Code in Services
  - prozedurale Programmmierung
  - blutleeres Modell
- zu viel Code in Entitäten
  - Überfrachtung der Entitäten mit Verhalten
  - Verlust konzeptioneller Klarheit
  - Abhängigkeiten an der falschen Stelle

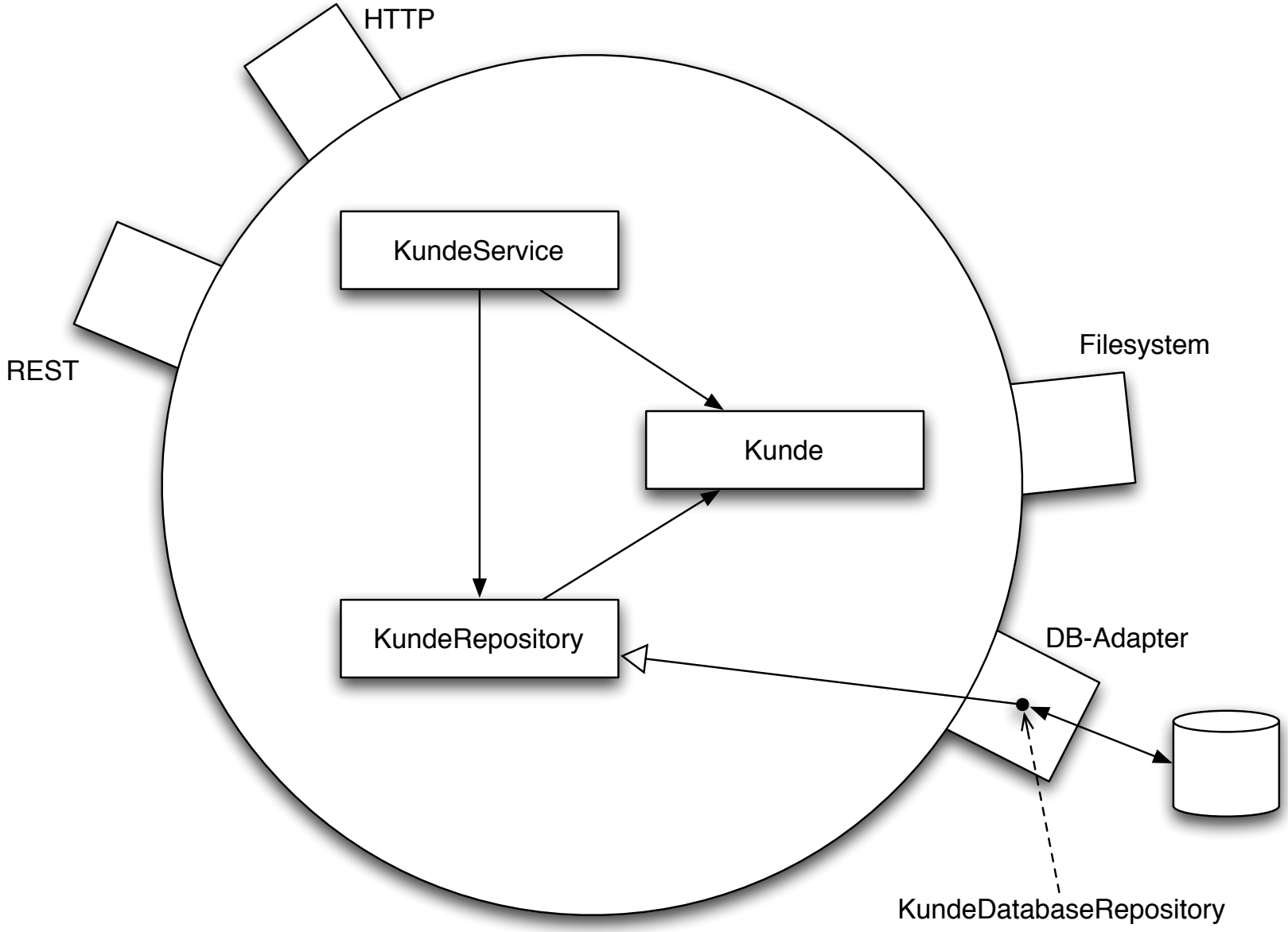
Repositories

- fachliche Schnittstelle zu Daten
- gaukeln In-Memory-Collection vor
- keine Infrastruktur-Abhängigkeiten in Domain Code

Repositories



Repositories



Gründe für Anpassungen am Modell

- Grund Nummer 1: Lernen
- bessere Begriffe
- Beziehungen werden klarer
- neue Konzepte/Abstraktionen tauchen auf

Und sie programmierten glücklich

bis an ihr Lebensende. . .

Achtung!

- Technisches nicht hinten runterfallen lassen
- Vorsicht vor blutleeren Modellen - Code in die Objekte!
- Services sollten die Ausnahme sein, nicht die Regel
- klassische Schichtenarchitektur kann suboptimal sein

Positive Aspekte

- Fokus auf Fachlichkeit und gemeinsamer Sprache
- gutes Zusammenspiel mit Specification by Example bzw. Acceptance Test-Driven Development (ATDD)
- Value Objects entlasten Entitäten und ziehen Code an
- keine Angst vor Veränderungen - Code geschmeidig halten

Vielen Dank!

Folien auf GitHub:

`https://github.com/NicoleRauch/DomainDrivenDesign`

Schulung bei Digicomp:

Domain-Driven Design - Lernen Sie Software mit optimalem Geschäftsnutzen zu entwickeln

Nicole Rauch

E-Mail `info@nicole-rauch.de`

Twitter `@NicoleRauch`

Arif Chughtai

E-Mail `mail@arifchughtai.org`