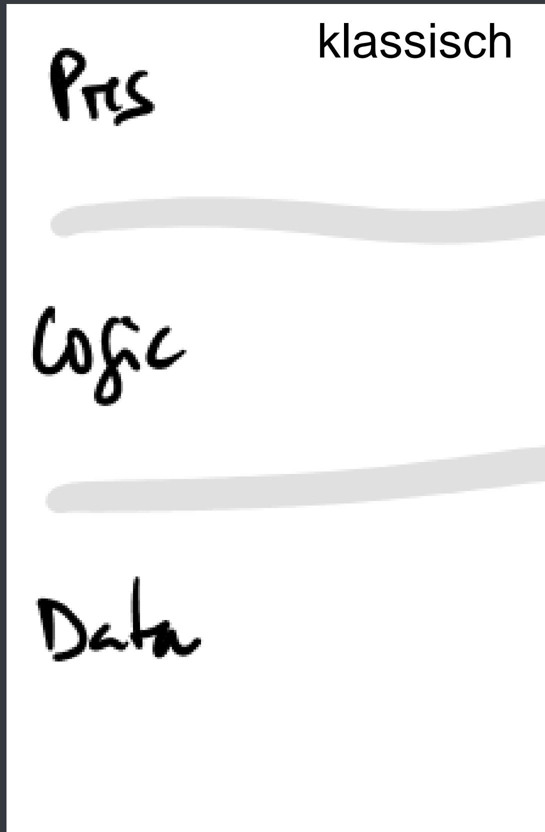
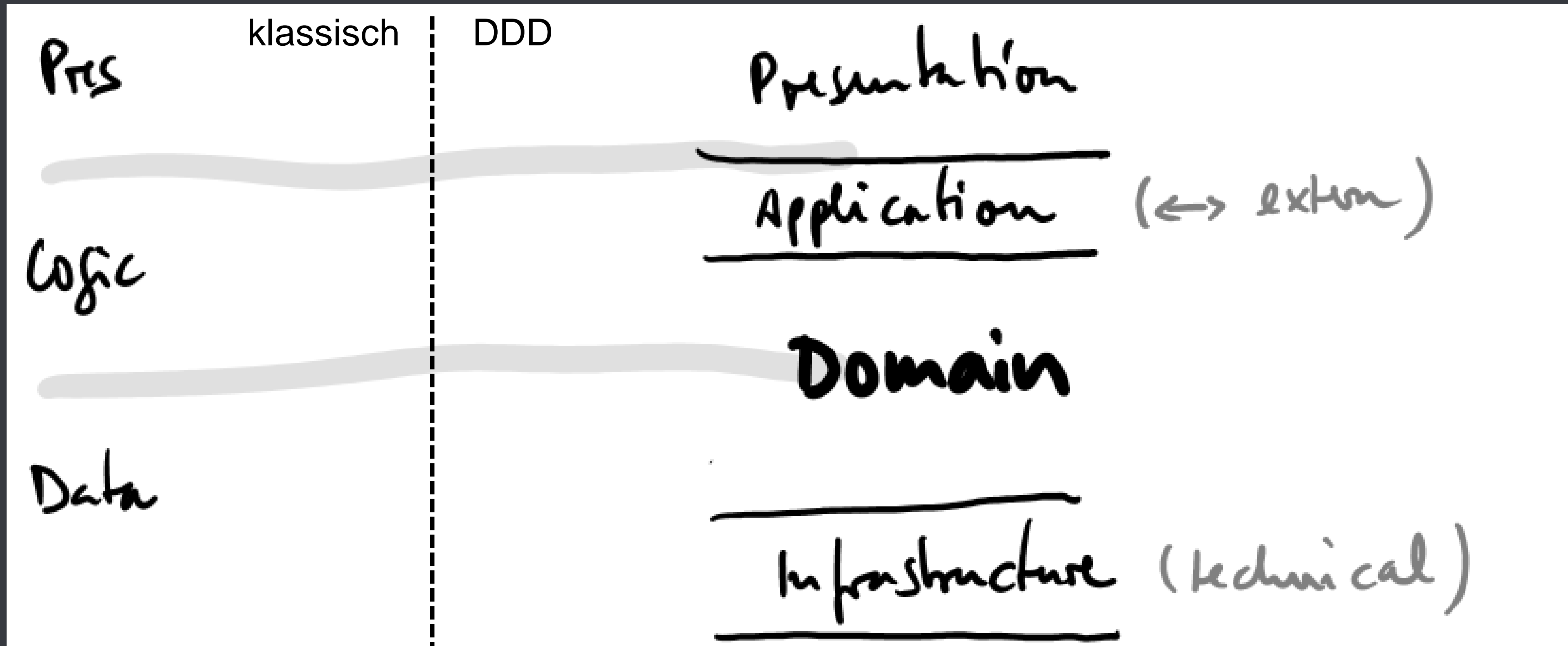
DDD-Schichtenmodell

Schicht	
Presentation Layer	User Interface, Nutzerinteraktion
Application Layer	Life Cycle, Orchestrierung, Prozesse, Kommunikation Außenwelt, „Use Cases“
Domain Layer	~~Technologie~~ Geschäftslogik
Infrastructure Layer	Technologie

Klassisches vs. DDD-Schichtenmodell



Klassisches vs. DDD-Schichtenmodell



Technologie-agnostischer Domain Layer

- \+ Technologiewechsel leicht
- \+ Übersichtlichkeit, Business-Logik „pur“
- \+ Testbarkeit
- \- Mehraufwand bei Ersterstellung
- \- ggfs. redundante Strukturen

Hexagonal Architecture

- Ports & Adapters statt Layers
  - =~ „technische“ (Infrastruktur-)
  - Aspekte des Application Layers
  - Adapter = explizite technische
  - Controller-Elemente

- Application und Domain-Kern sonst ähnlich wie DDD

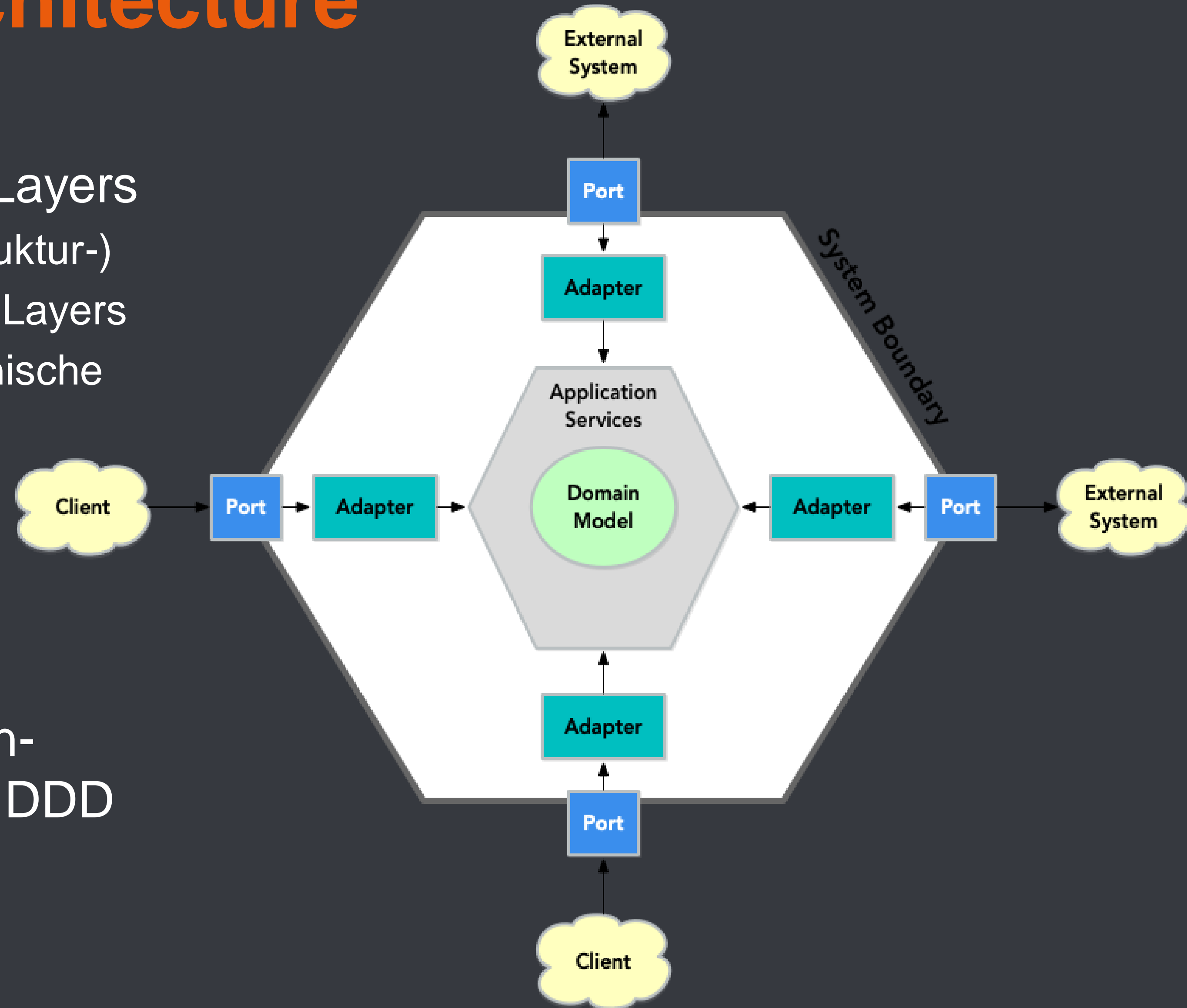


Bild: Holmström, P. (2020). *Domain-Driven Design and the Hexagonal Architecture.* Vaadin. https://vaadin.com/learn/tutorials/ddd/ddd_and_hexagonal

Hexagonal & Onion Architecture

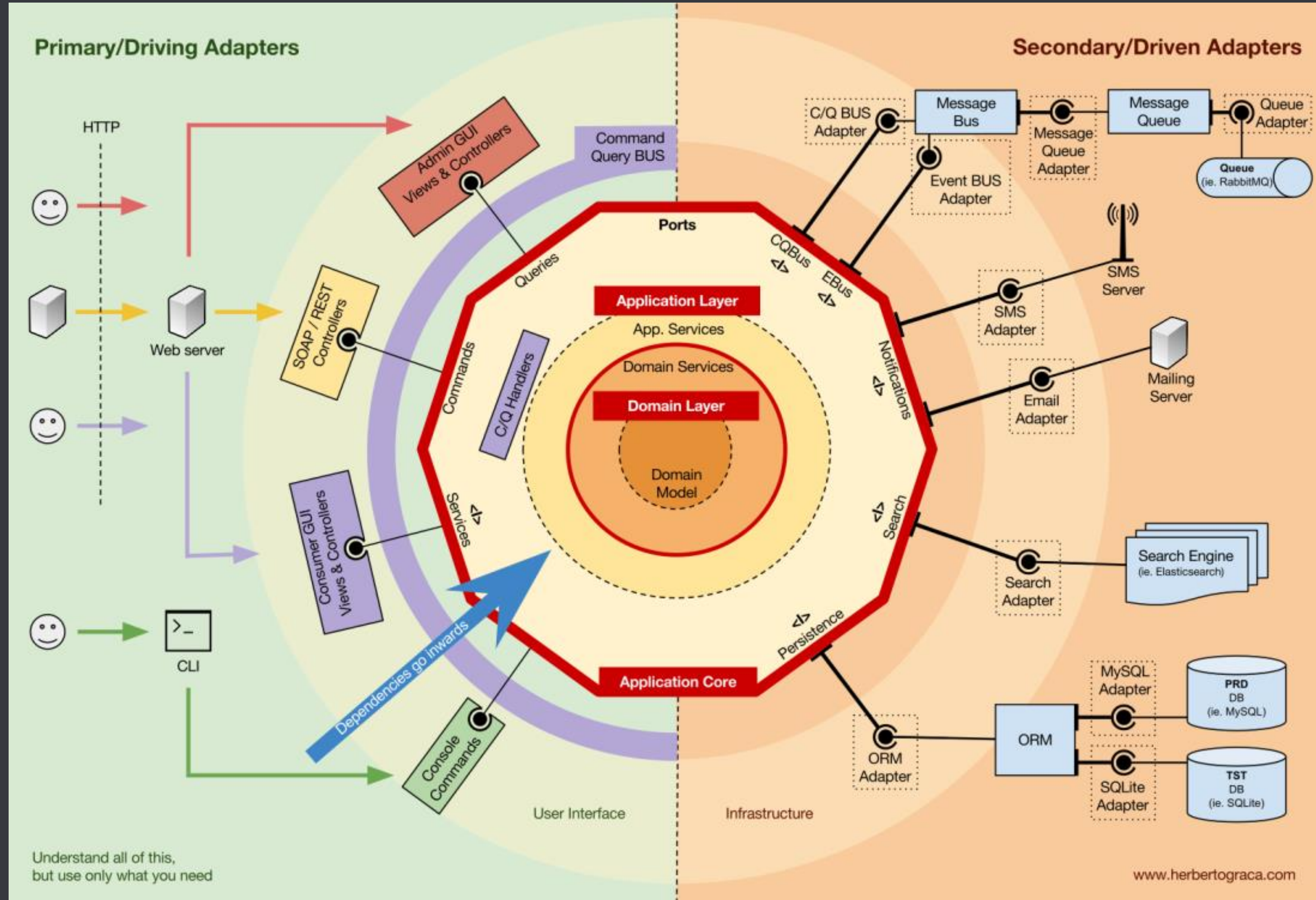
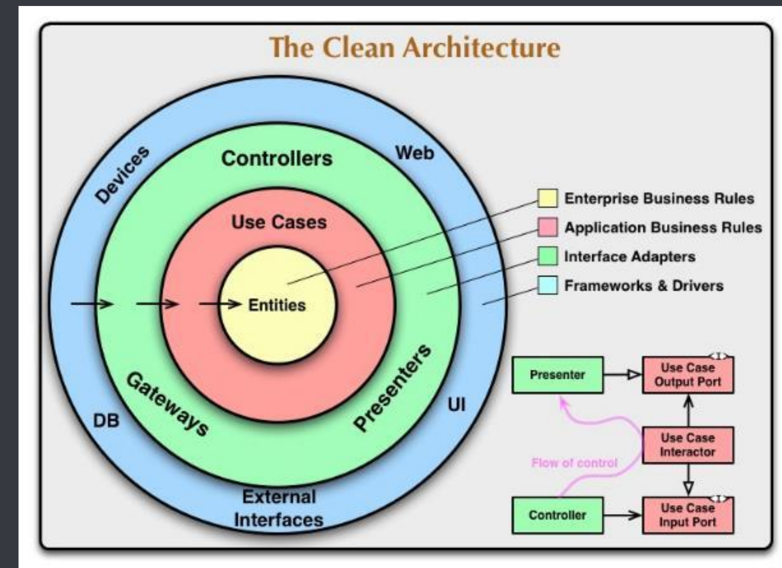
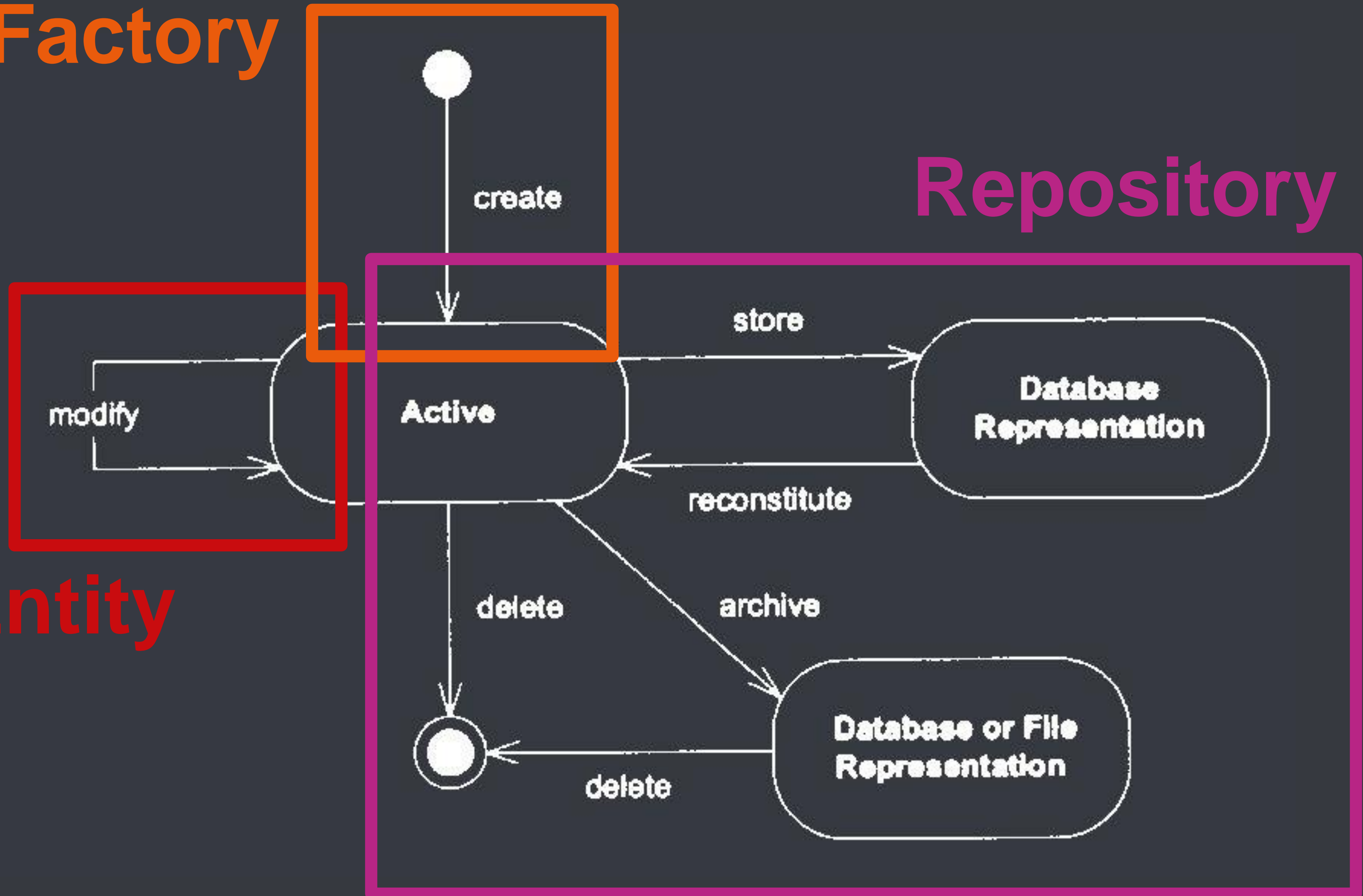


Bild: Graca, H. (2017, November 16). DDD, Hexagonal, Onion, Clean, CQRS, … How I put it all together. *@hgraca.* https://herbertograca.com/2017/11/16/explicit-architecture-01-ddd-hexagonal-onion-clean-cqrs-how-i-put-it-all-together/

Clean Architecture (Bob Martin)

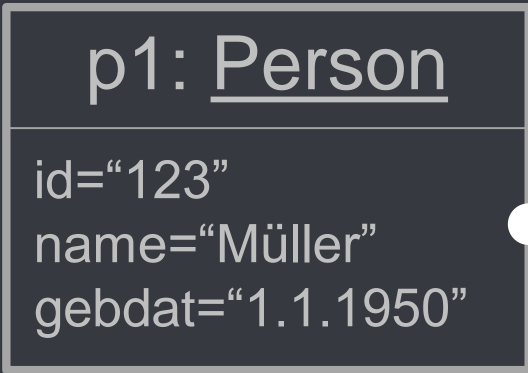
- Clean Architecture = ähnliche ringförmige Struktur wie Hexagonal Architecture und Onion Architecture
- Vergleich zu DDD:
  - **Äußerer blauer** Ring = (wie bei HA und OA) Mischung aus Presentation und Infrastructure Layer
  - Der **grüne** und der **rote** Ring entsprechen dem Application Layer
    - Use Cases entsprechen Application Layer Services
  - Der **gelbe** Kern entspricht dem Domain Layer

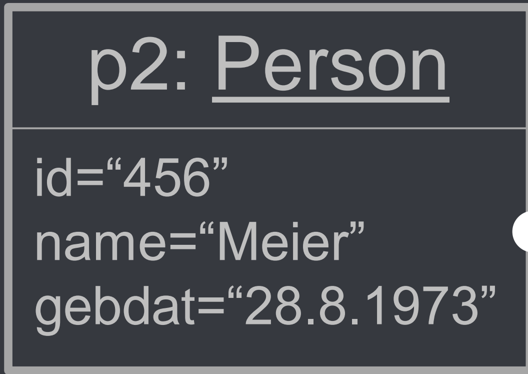



Factory
Repository
Entity
create
store
modify
Active
Database
Representation
reconstitute
delete
archive
Database or File
Representation
delete

Warum Datenbanken für persistente Speicherung?

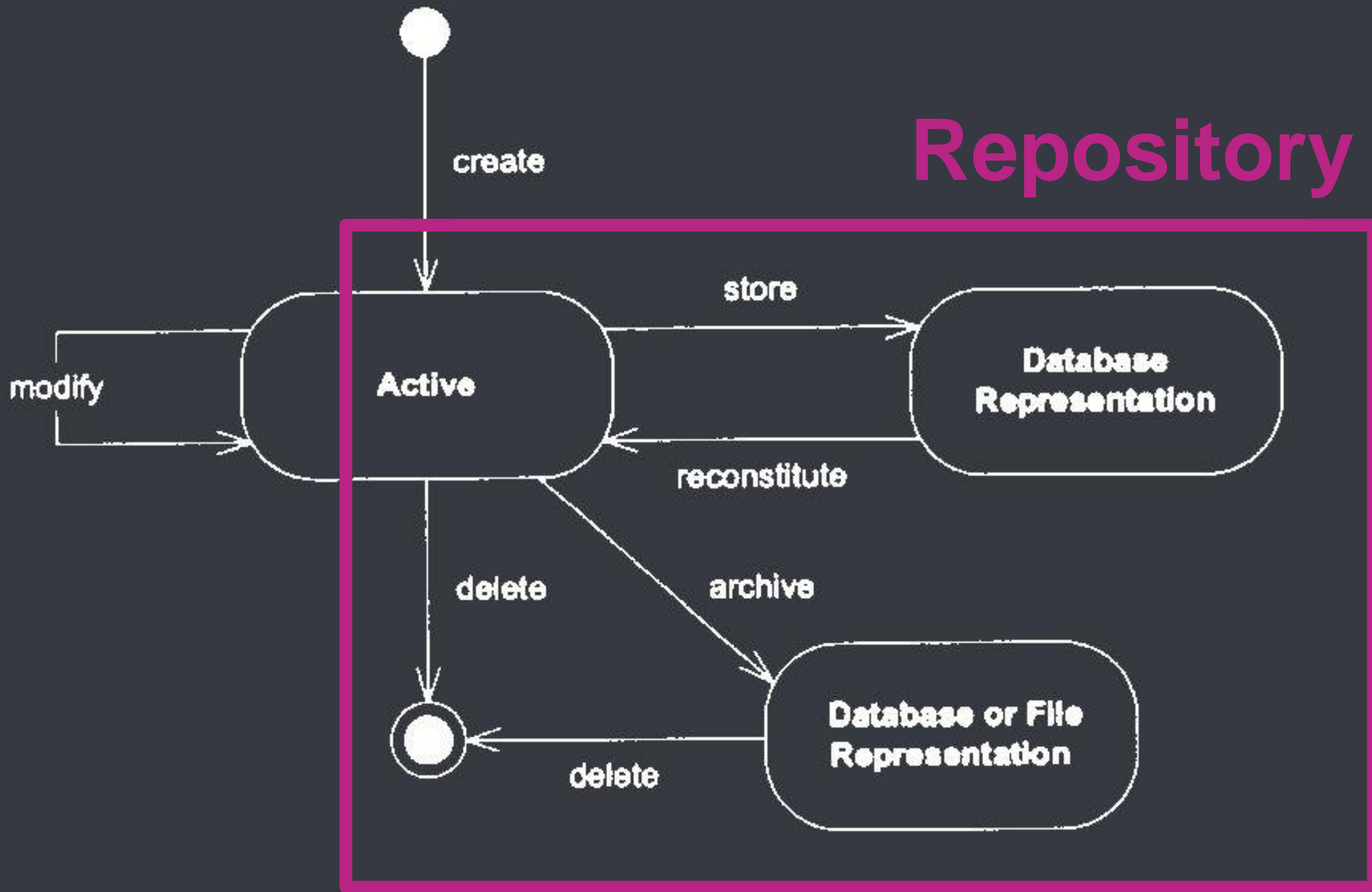
1. Redundanzfreiheit
2. Konsistenz
3. Zuverlässigkeit
4. Abfragen
5. Rollen- / Rechte-Konzept
6. Mehrbenutzerbetrieb
7. Transaktionskonzept





id	name	gebdat

DBTable PERSON


Repository
create
store
modify
Active
Database
Representation
reconstitute
delete
archive
Database or File
Representation
delete

Create

```
Kunde k = new Kunde( "Meier" );
someRepository.save( k );
// Tabellenzeile für k
// in der Datenbank angelegt
```

Read

```
k = someRepository.
        findByName( "Meier" );
// k wird in DB gesucht und
// ausgelesen
```

```
@Entity
public class
Kunde {
    …
}
```

Update

```
k.setName( "Müller" );
someRepository.save( k );
// Werte für k in DB geändert
```

Delete

```
someRepository.delete( k );
// Zeile für k in DB gelöscht
```

KUNDE
ID: int
KUNDEN_NR: varchar(10)
VORNAME: varchar(30)
NACHNAME: varchar(30)

Value Objects mit EF Core

- Einfach

```
public struct …
```

Komplex

```
[ComplexType]
[Owned]
public class …
```

- keine public Setter

```
public int Number { get; private set; }
```

- Equals / Hash mit allen Attributen

Entities mit EF Core

- ID (Guid besser als int!)

```
[Key]
[DatabaseGenerated(DatabaseGeneratedOption.Identity)]
public Guid Id { get; private set; }
```

- Non-nullable Properties

```
[Required]
```

- Beziehungen
  - implizit oder mit Fluent API (dazu später mehr)

Repositories mit EF Core

- Abgeleitet von `DbContext`
- Siehe Code
- Beispiel-Repo *(work in progress)* https://gitlab.com/archi-lab/public/digital-exams-with-persistance

www.archi-lab.io

Videoserie „Softwaretechnik“ - © 2020 Prof. Dr. Stefan Bente