ACCSO

Moderne Softwarearchitekturen im DDD-Umfeld – von Hexagonaler über Onion bis Clean Architektur

Unterschiede und Gemeinsamkeiten

Gulmariyam Yerzhanova

Gulmariyam Yerzhanova

- Software Engineer bei der Accso GmbH
- Leidenschaft für den Bereich Softwarearchitektur und Domain-Driven Design

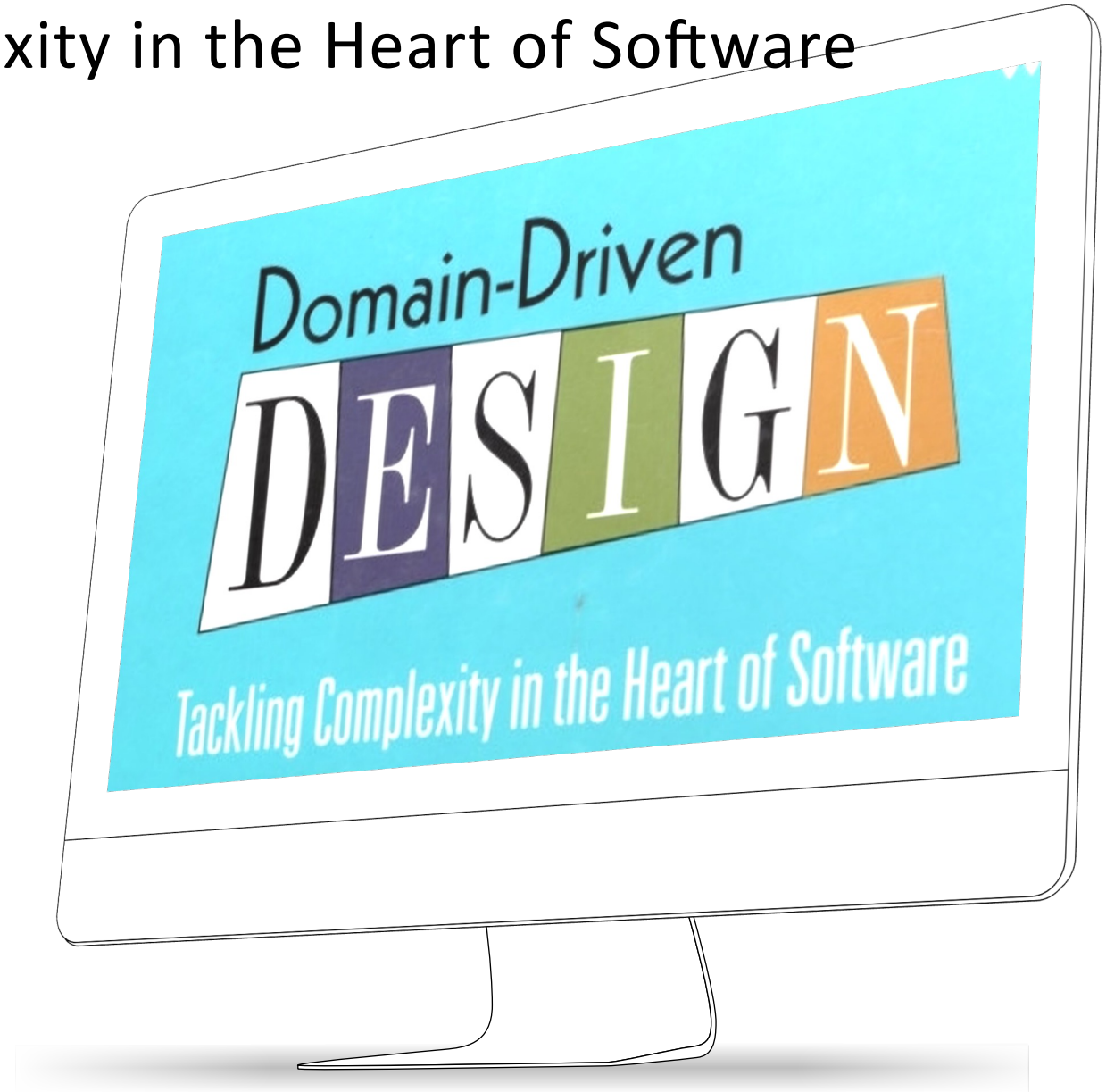
Agenda

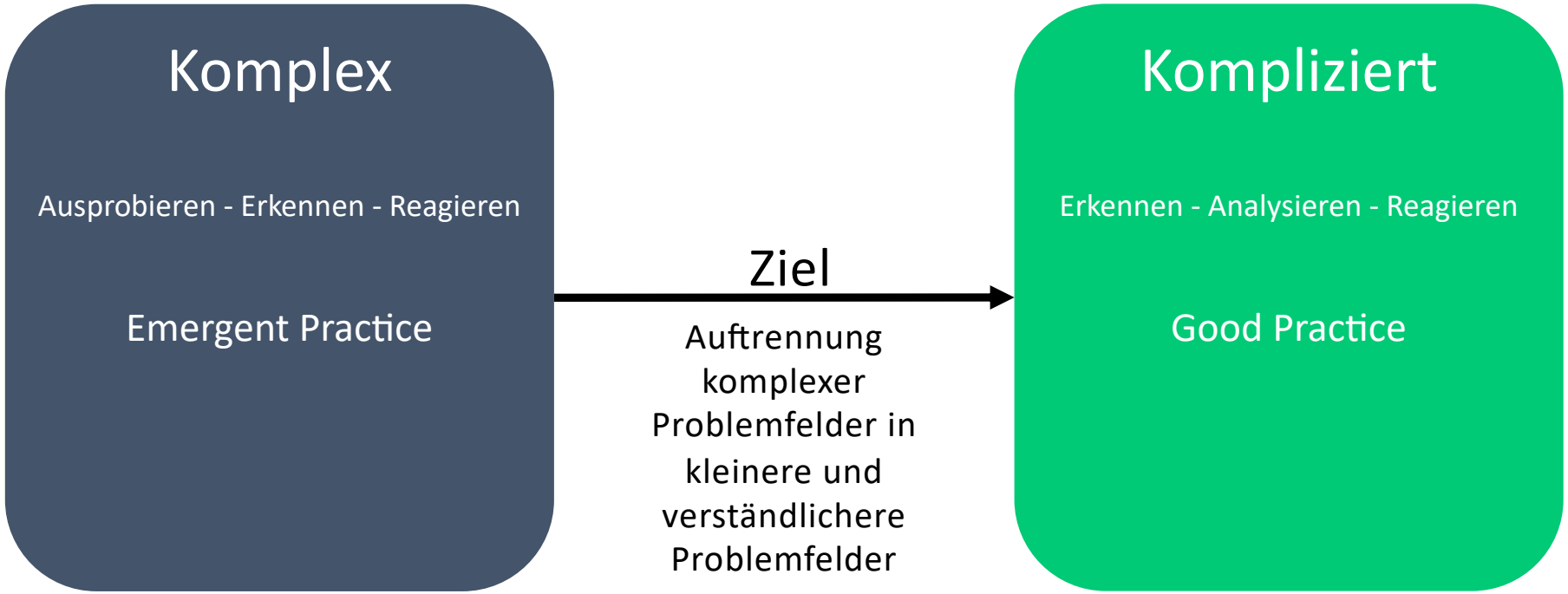
ACCSO

- Domain-driven Design
- Domänenzentrische Architekturen: Hexagonal, Onion, Clean
- Domain Model Implementierung

DDD - Tackling Complexity in the Heart of Software

Eric Evans

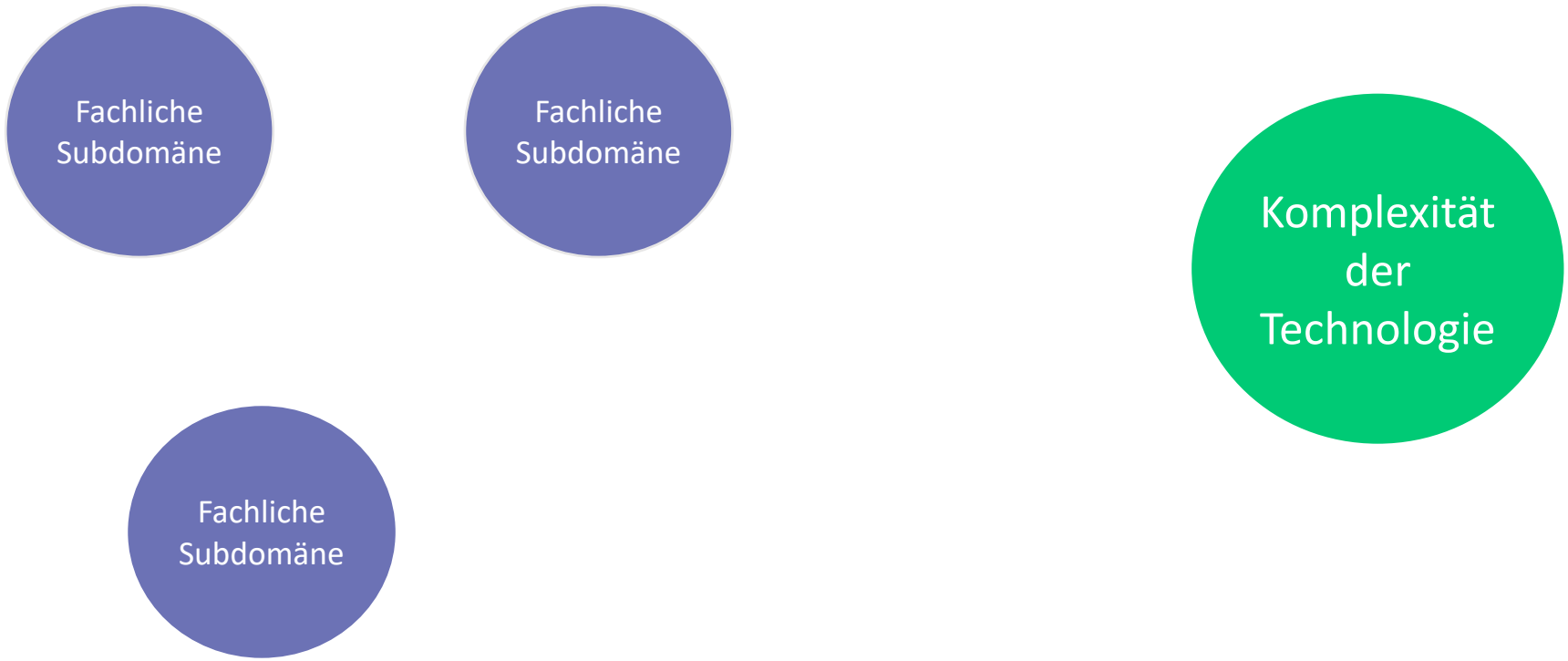



Komplex
Ausprobieren - Erkennen - Reagieren
Emergent Practice
Ziel
Auftrennung komplexer Problemfelder in kleinere und verständlichere Problemfelder
Kompliziert
Erkennen - Analysieren - Reagieren
Good Practice

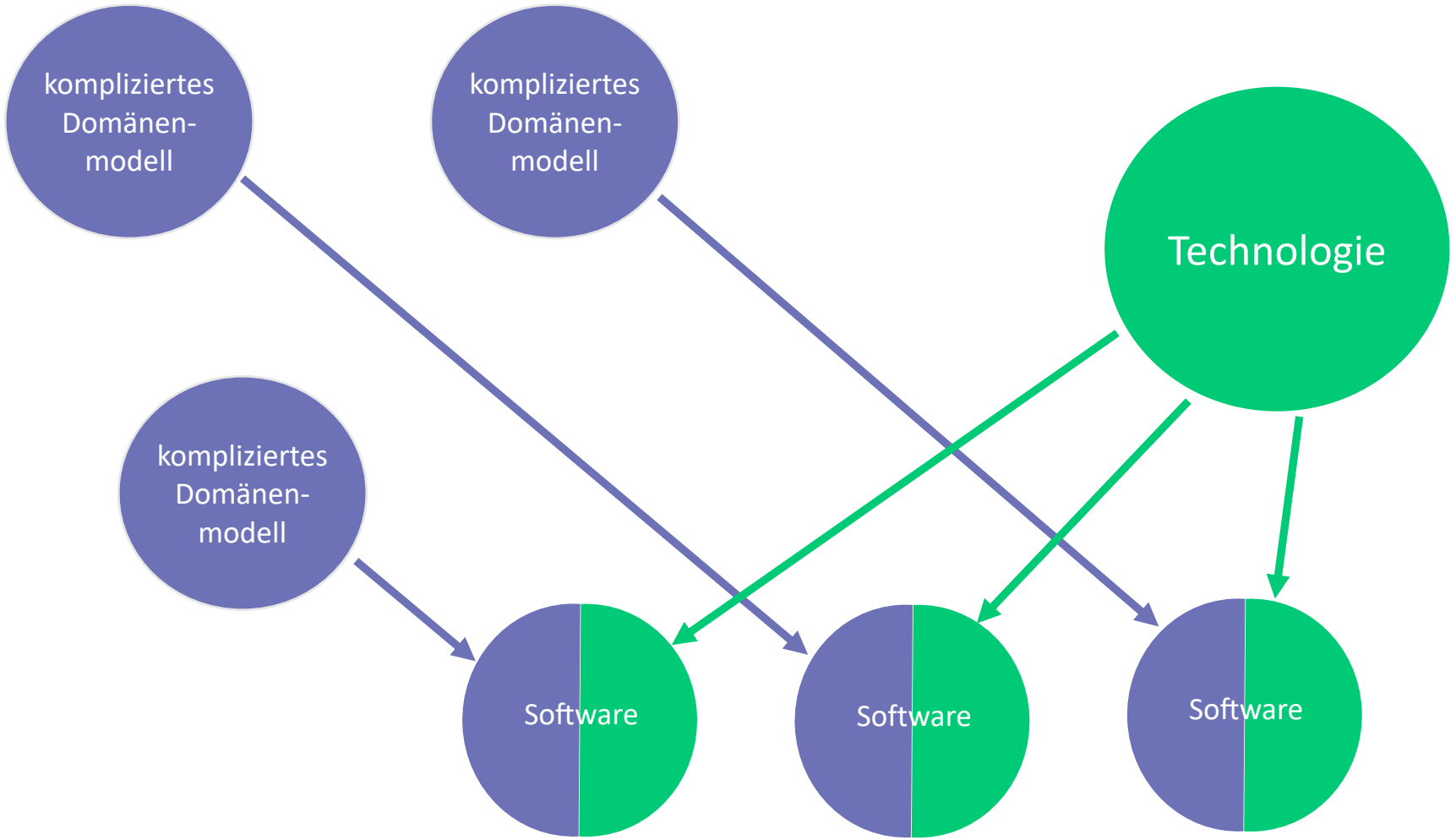

Komplexität
der
Software


Fachliche
Komplexität
der Domäne

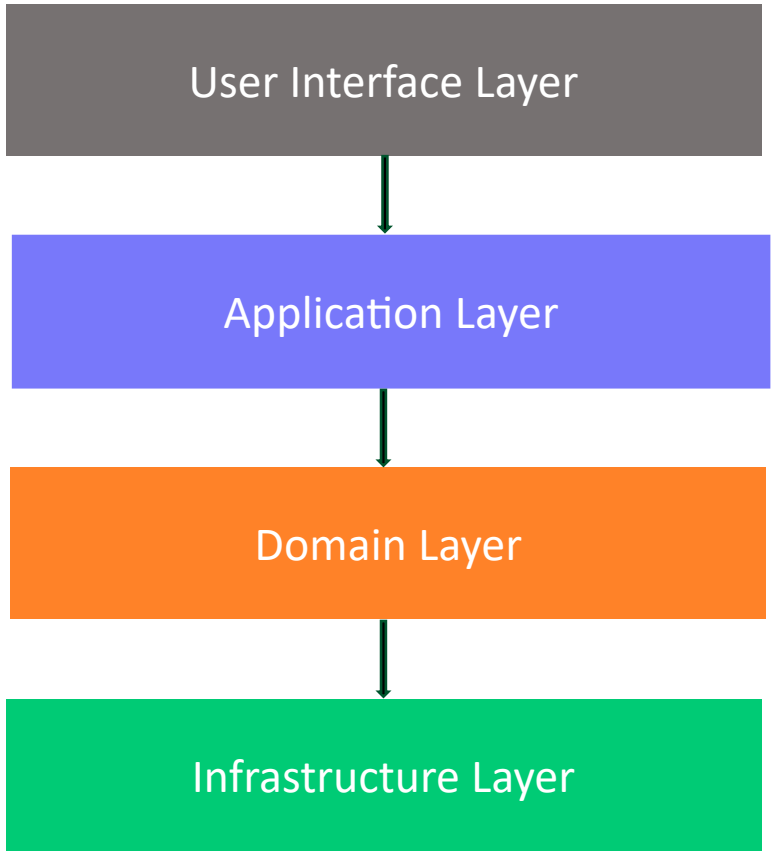

Komplexität
der
Technologie


Fachliche Subdomäne
Fachliche Subdomäne
Fachliche Subdomäne
Komplexität der Technologie


kompliziertes Domänen-modell
kompliziertes Domänen-modell
kompliziertes Domänen-modell
Komplexität der Technologie

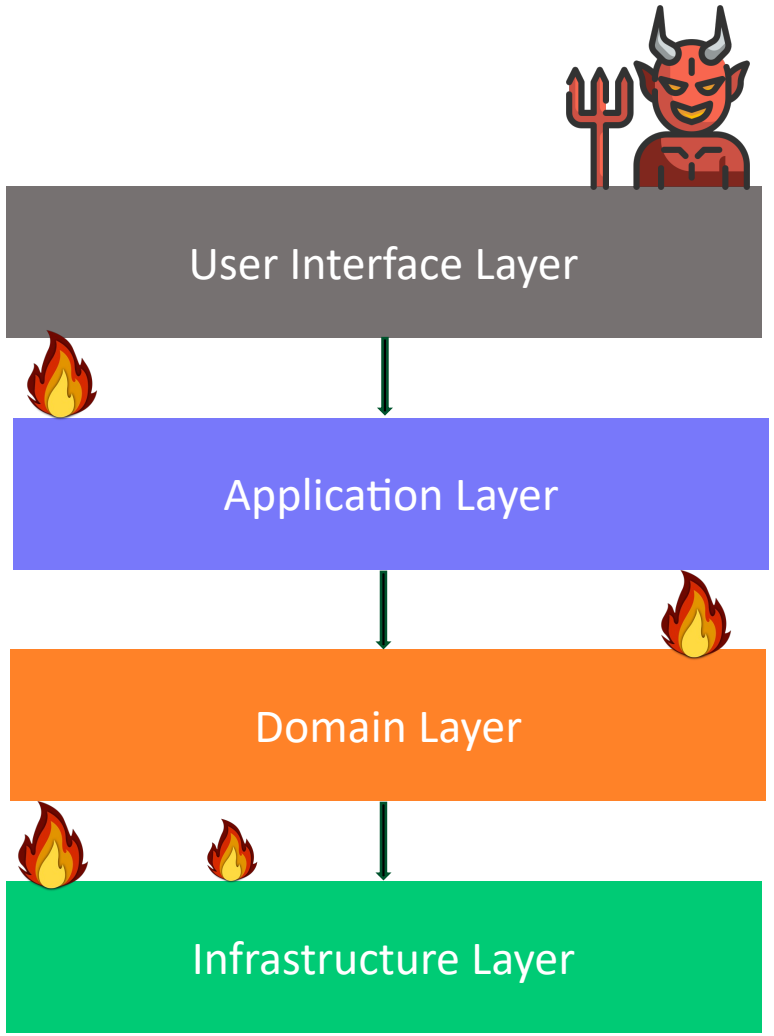

kompliziertes Domänen-modell
kompliziertes Domänen-modell
kompliziertes Domänen-modell
Technologie
Software
Software
Software

Schichtenarchitektur



Eine grundlegende Regel dieses Architekturmusters ist, dass jede Schicht nur von sich selbst und den darunter liegenden Schichten abhängig sein sollte.

Schichtenarchitektur



Interfaces und Implementierungen

Accso

Interfaces und Implementierungen

DEPENDENCY INVERSION PRINZIP

Accso

Dependency Inversion Principle

The Dependency Inversion Principle (DIP) states that high level modules should not depend on low level modules; both should depend on abstractions. Abstractions should not depend on details. Details should depend upon abstractions.

Entstehungsgeschichte domänenzentrischer Architekturen

2005

Hexagonale Architektur
Alistair Cockburn

Entstehungsgeschichte domänenzentrischer Architekturen

2008

Onion Architektur

Jeffrey Palermo

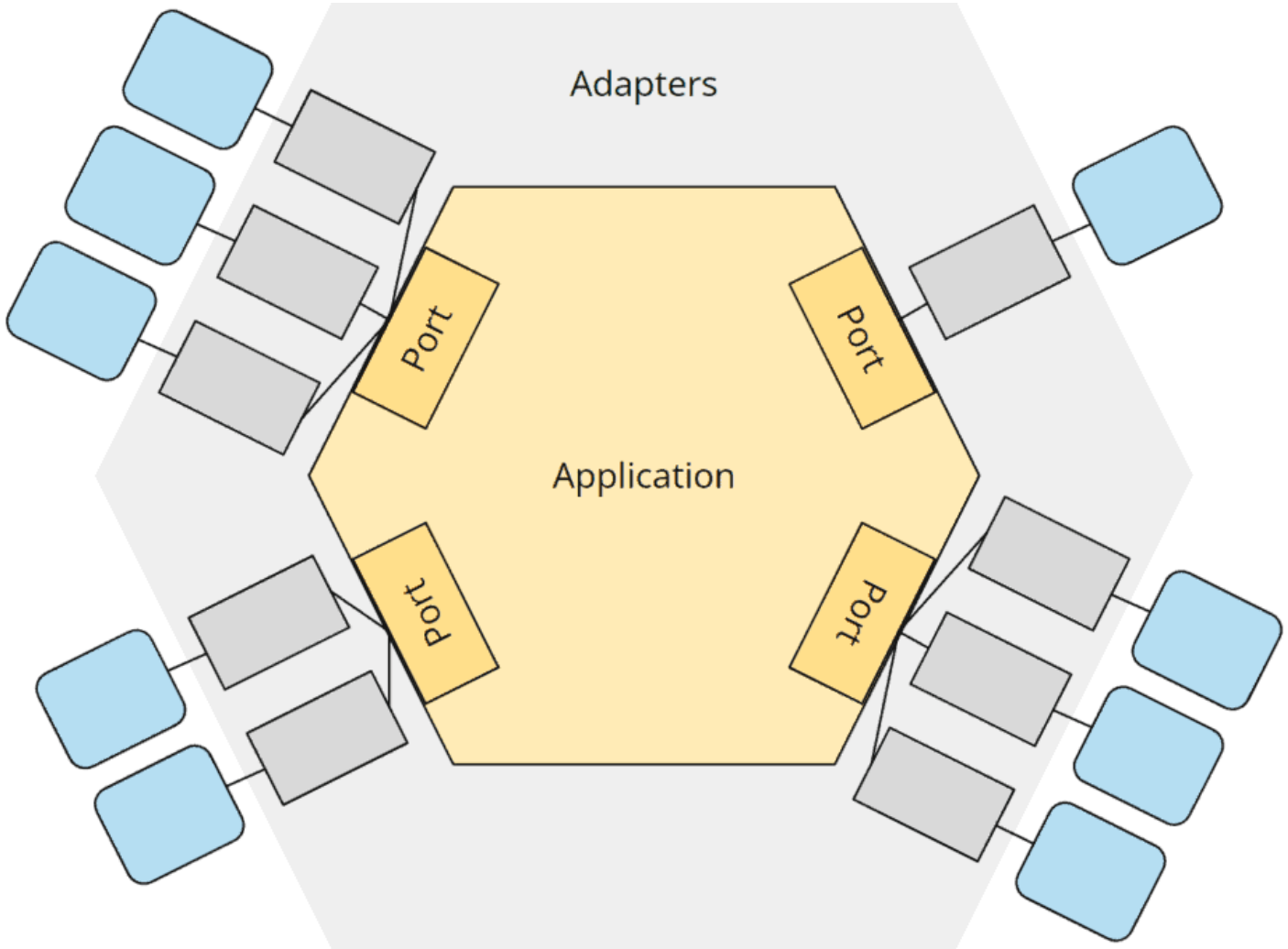
Entstehungsgeschichte domänenzentrischer Architekturen

2018

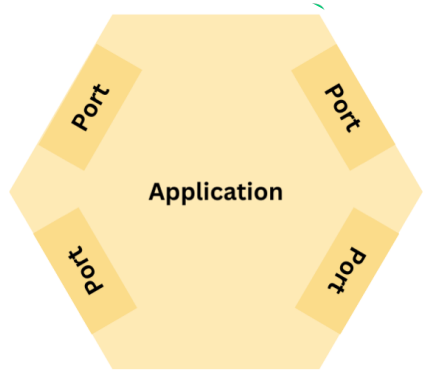
Clean Architektur
Robert C. Martin

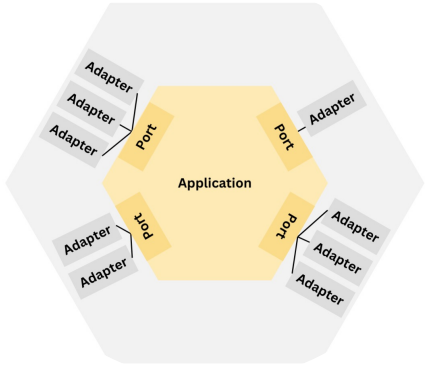
Hexagonale Architektur
Alistair Cockburn

Hexagonale Architektur



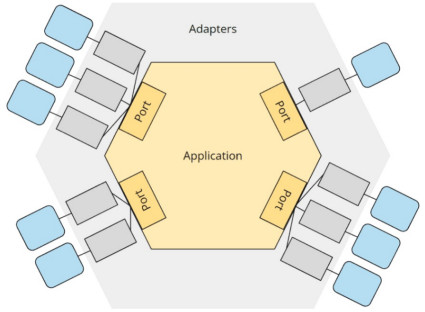
Ports





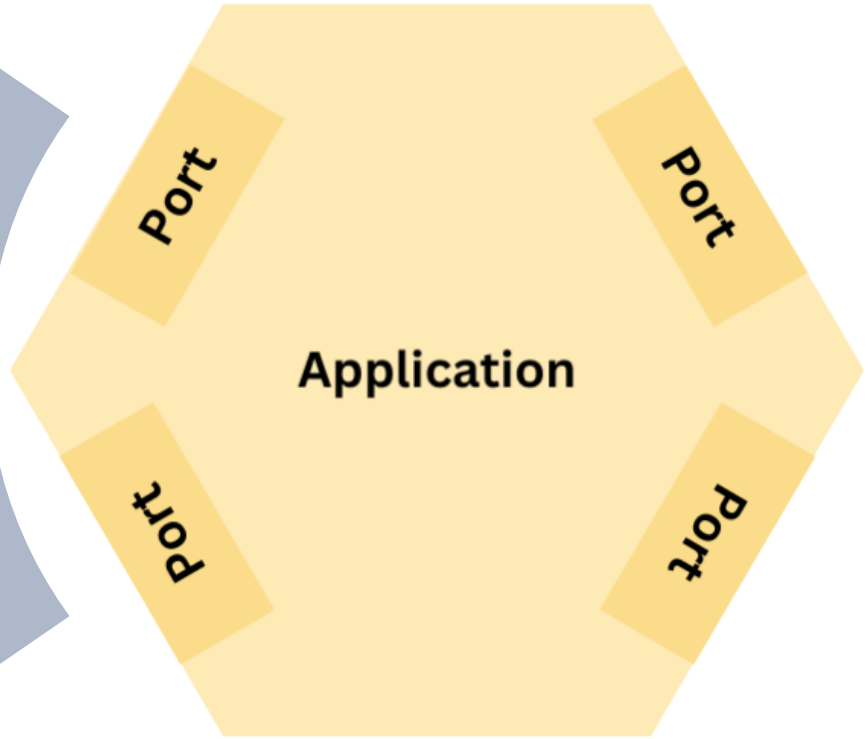
Adapters

Application

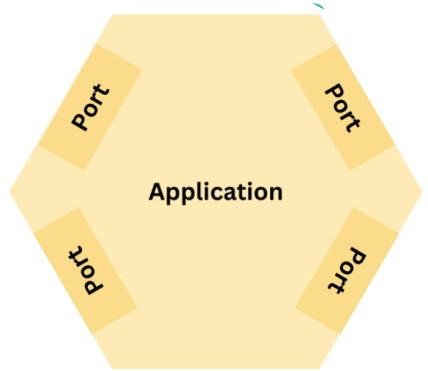


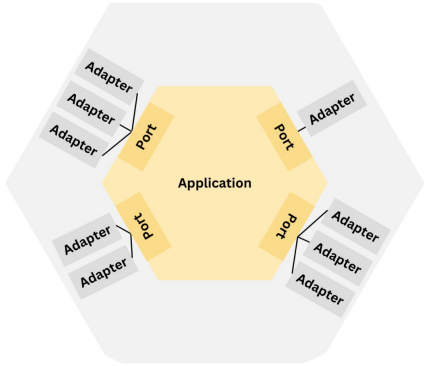
Ports

- Ports = Schnittstellen, die Geschäftslogik definieren, um mit der Außenwelt(Fremdsysteme, DB usw.) zu interagieren
- Geschäftslogik-Implementierungen gemäß den definierten Ports



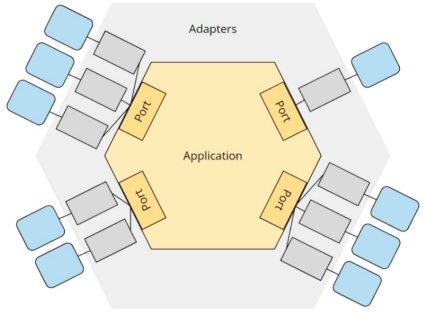
Ports



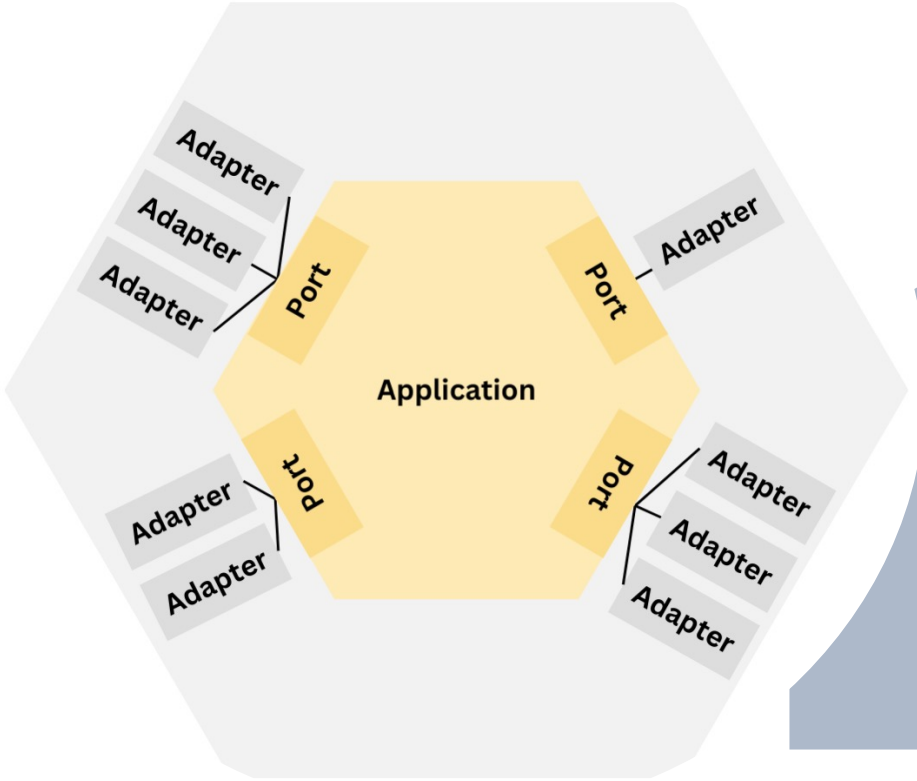


Adapters

Application

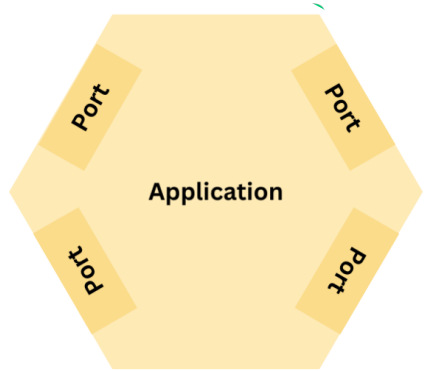


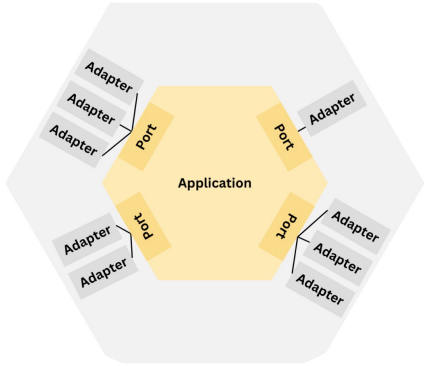
Adapters



- Adapters = Realisierung der Schnittstellen (Ports), um mit der Anwendung zu kommunizieren
- Adapter ermöglichen Integration verschiedener Technologien
- Anpassung von Technologien ohne Beeinträchtigung der Kernlogik

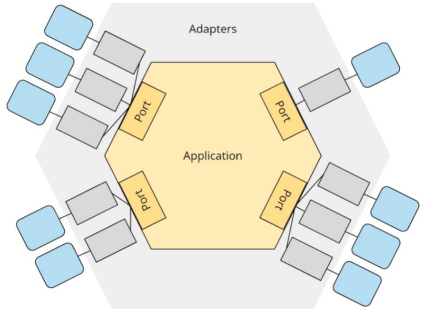
Ports

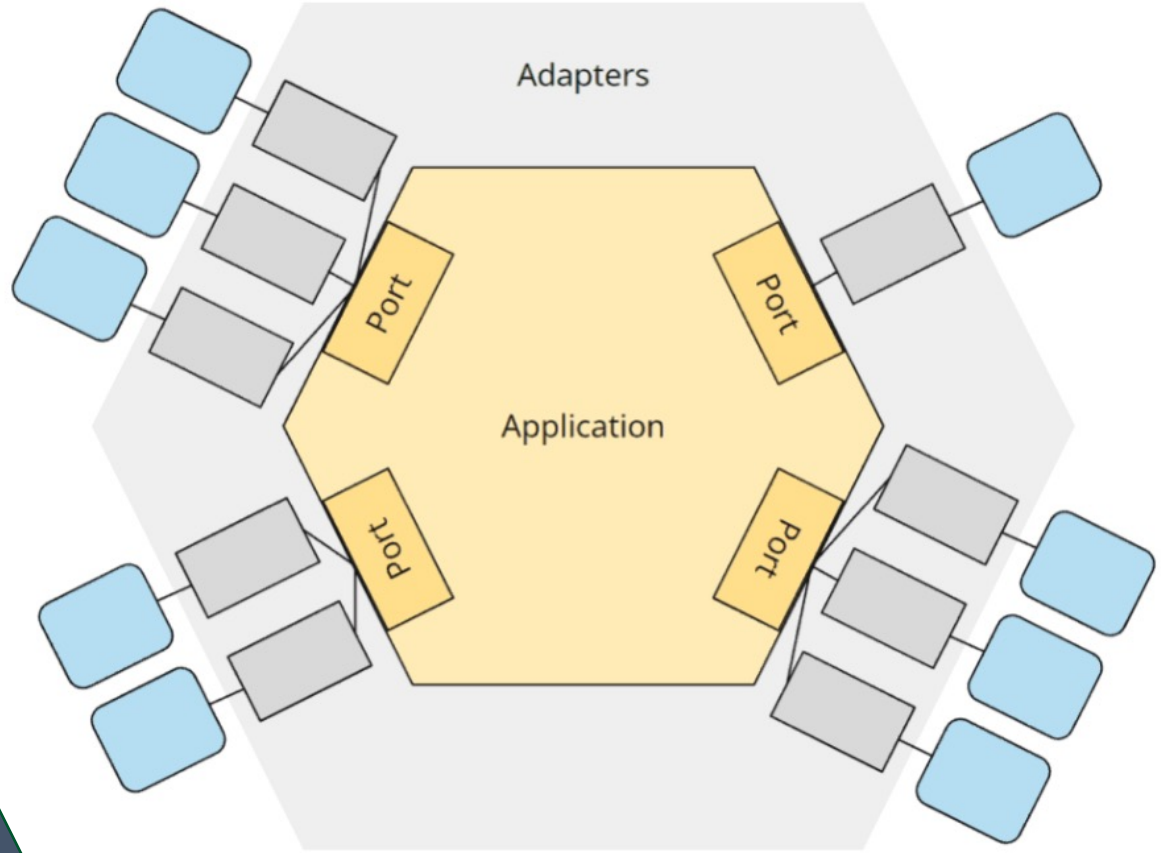




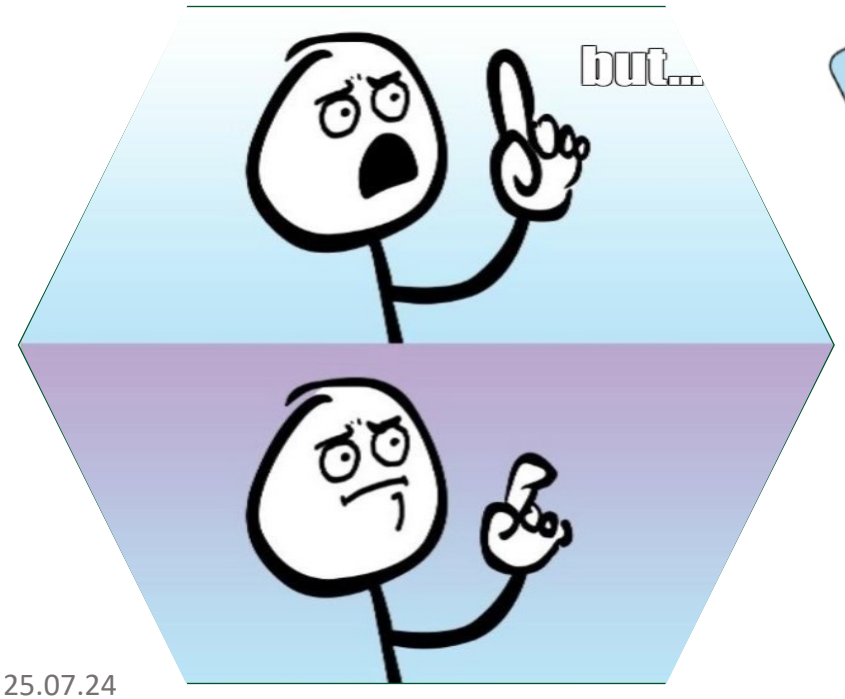
Adapters

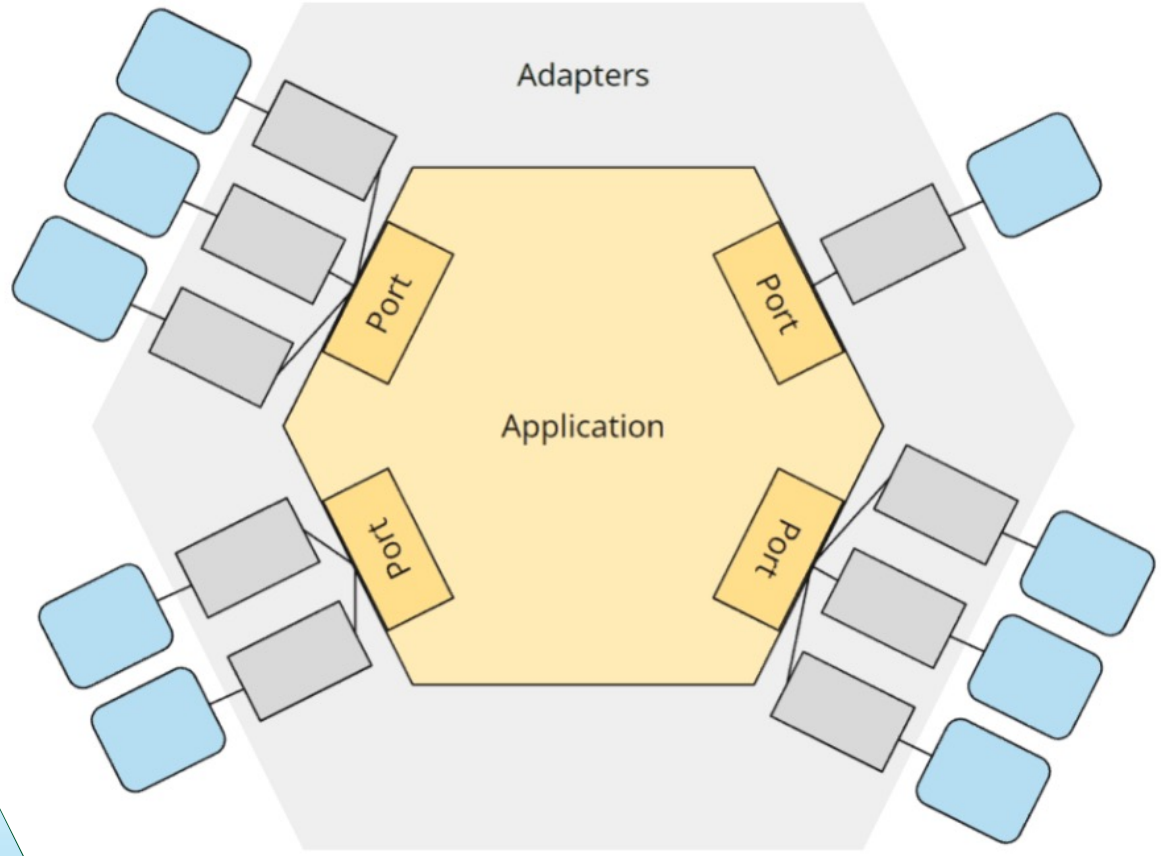
Application





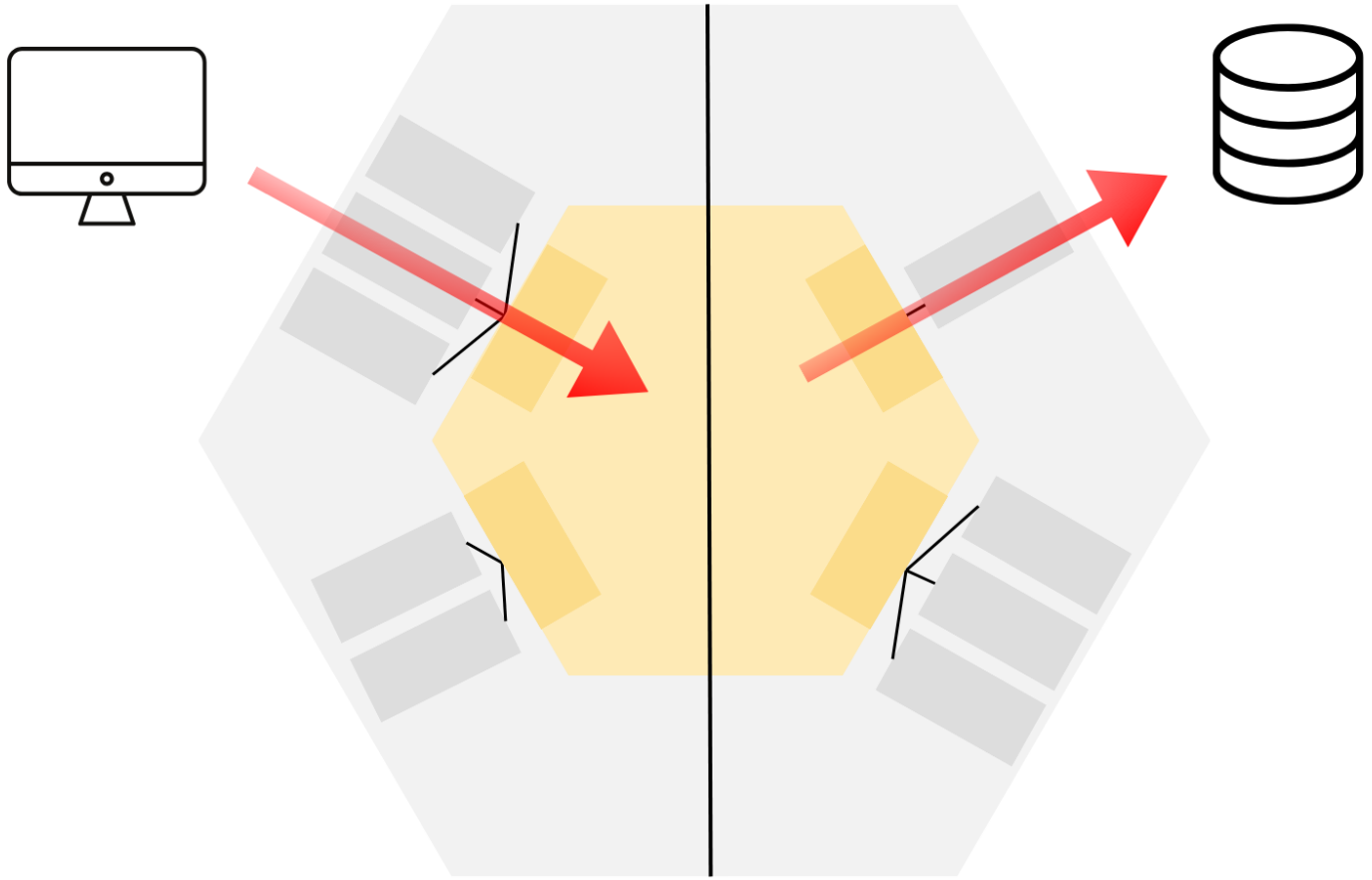
Bei der Beschreibung des Musters ”Ports und Adapter” wird bewusst angenommen, dass alle Ports und Adapters grundsätzlich ähnlich sind.


but...


Adapters
Port
Port
Application
Port
Port

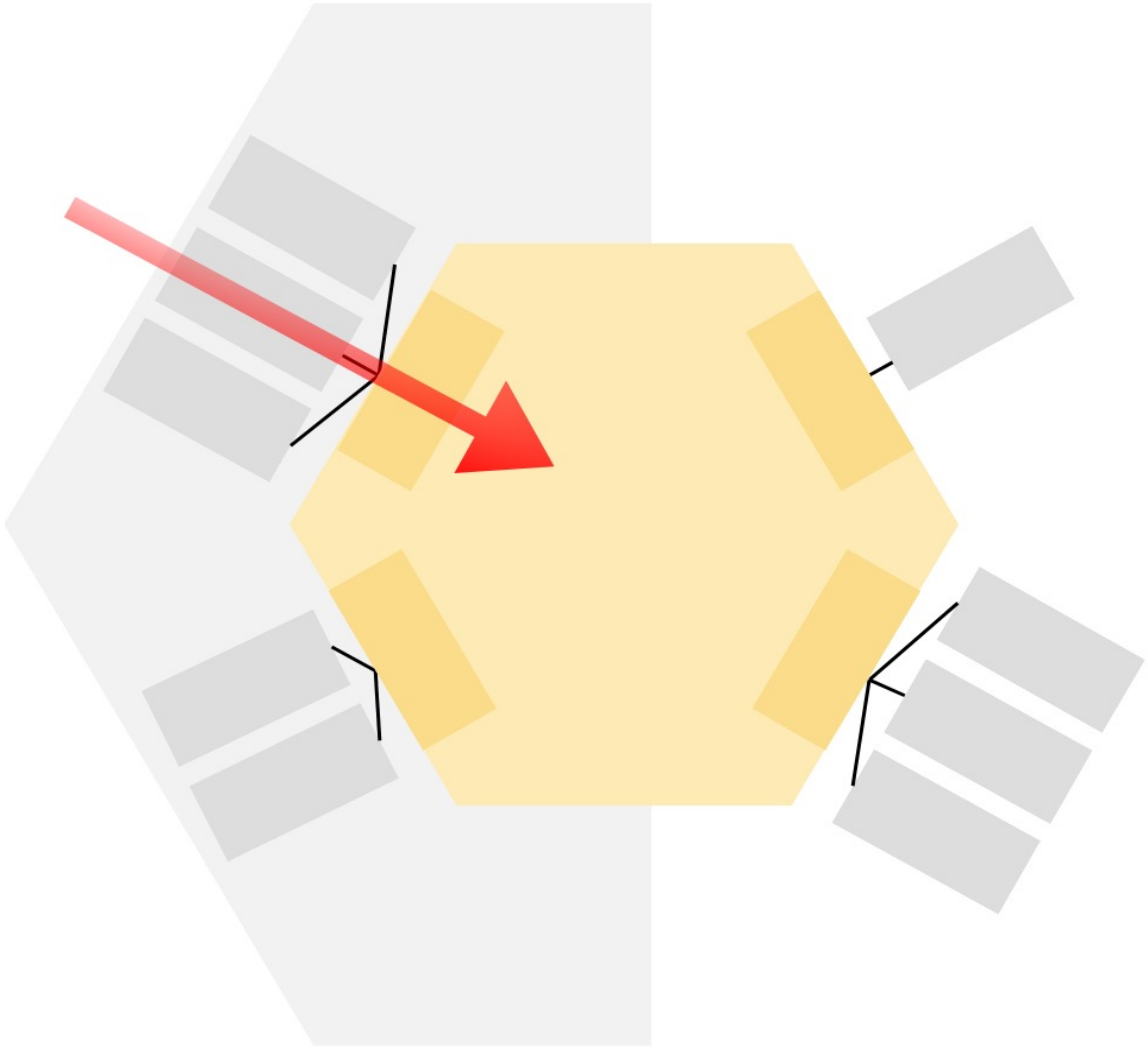
Input und Output Ports und Adapters

Accso



Input Ports und Adapters

- Solche, die die Anwendung treiben -> driving
- Übertragung/Bearbeitung von eingehenden Anfragen



Input Port

Accso

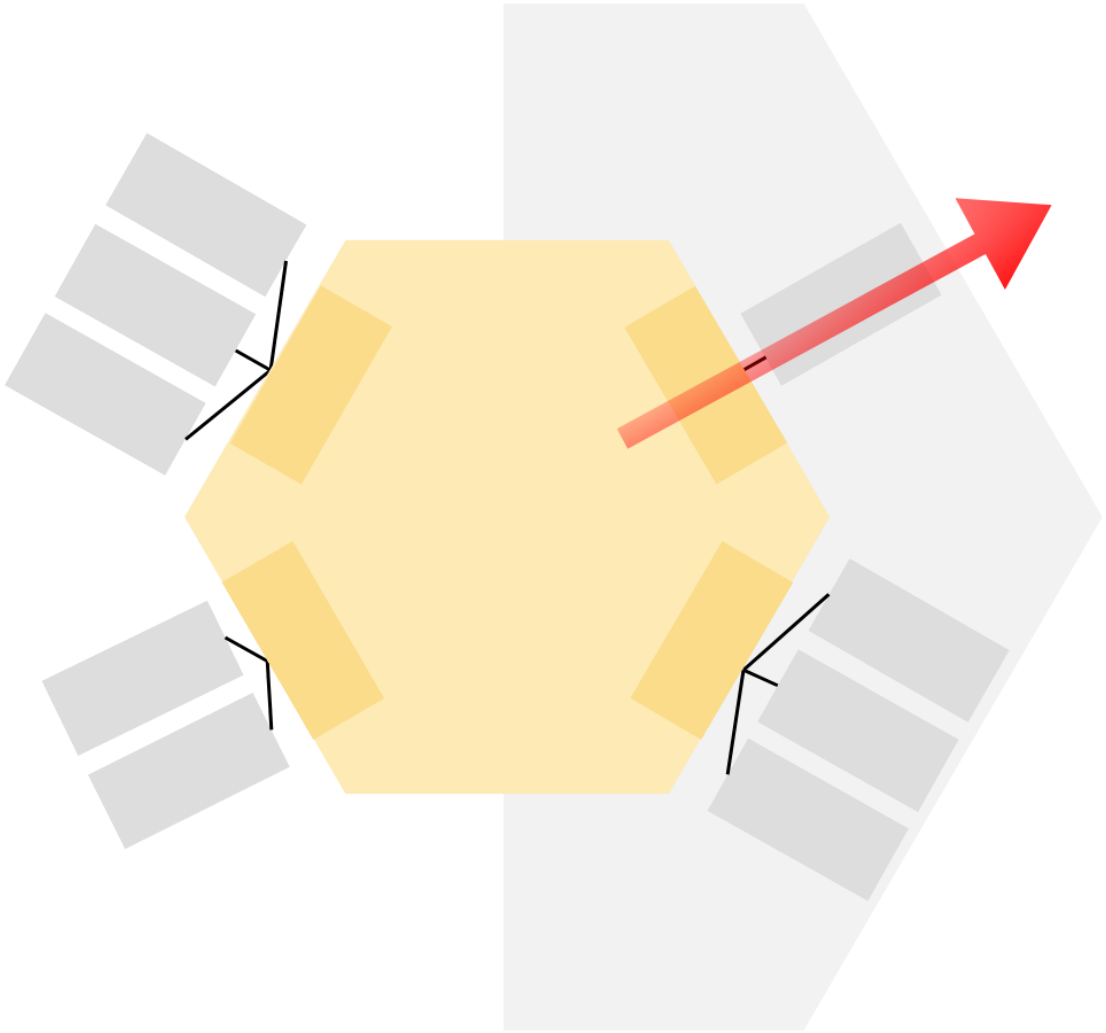
```
@Service
public interface LegeArtikelInDenWarenkorbUseCase {

    void legeArtikelInDenWarenkorb(UUID artikelId, int anzahl, UUID warenkorbID);

}
```

Input Adapter

Accso

```
@PostMapping("/neuerWarenkorb")
 public ResponseEntity<String> erstelleWarenkorbFuerKunde(HttpSession session,
                                                          @RequestParam String warenkorbID,
                                                          @RequestParam String kundeID,
                                                          @RequestParam BigDecimal maxEinkaufswert) {
    try {
        UUID warenkorb = UUID.fromString(warenkorbID);
        UUID kunde = UUID.fromString(kundeID);

        erstelleWarenkorbFuerKundeUseCase.erstelleWarenkorbFuerKunde(warenkorb, kunde, maxEinkaufswert);

        session.setAttribute("warenkorbID", warenkorb);
        return new ResponseEntity<>("Der Warenkorb wurde erfolgreich erstellt", HttpStatus.OK);
    } catch (IllegalArgumentException exception) {
        return new ResponseEntity<>("Fehler beim Erstellen des Warenkorbs",
                                    HttpStatus.INTERNAL_SERVER_ERROR);
    }
}
```

Output Ports und Adapters



- Solche, die von der Anwendung getrieben werden -> driven
- Bearbeitung von ausgehenden Anfragen

Output Port

Accso

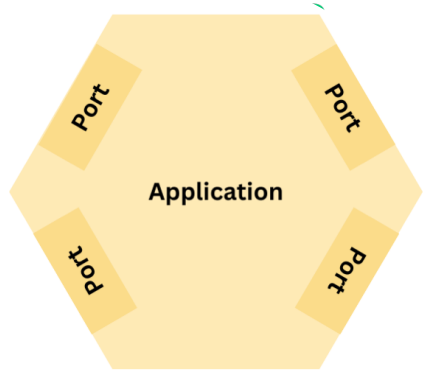
```
@Repository
public interface WarenkorbRepository {

    Warenkorb findeMit(WarenkorbID warenkorbId);
    void speichere(Warenkorb warenkorb);
    void entferne(WarenkorbID warenkorbId);

}
```

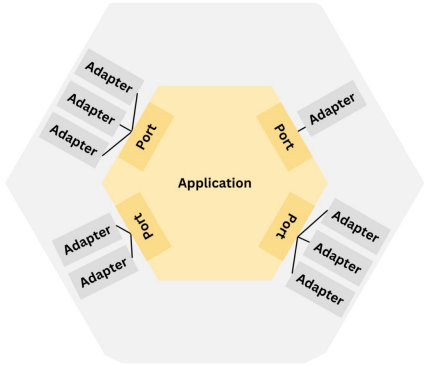
Output Adapter

Accso

```
public class WarenkorbRepositoryImplementation implements WarenkorbRepository {

    private final WarenkorbDAO warenkorbDAO;

    // ...

    @Override
    public Warenkorb findeMit(WarenkorbID warenkorbId) {
        WarenkorbDTO warenkorbDTO = warenkorbDAO.read(warenkorbId.warenkorbID());
        return WarenkorbDTOMapper.INSTANCE.vonDTOZuWarenkorb(warenkorbDTO);
    }

    // ...
}
```

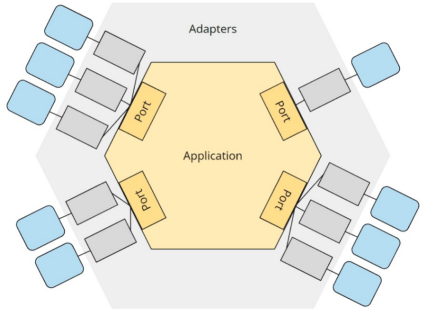
Ports



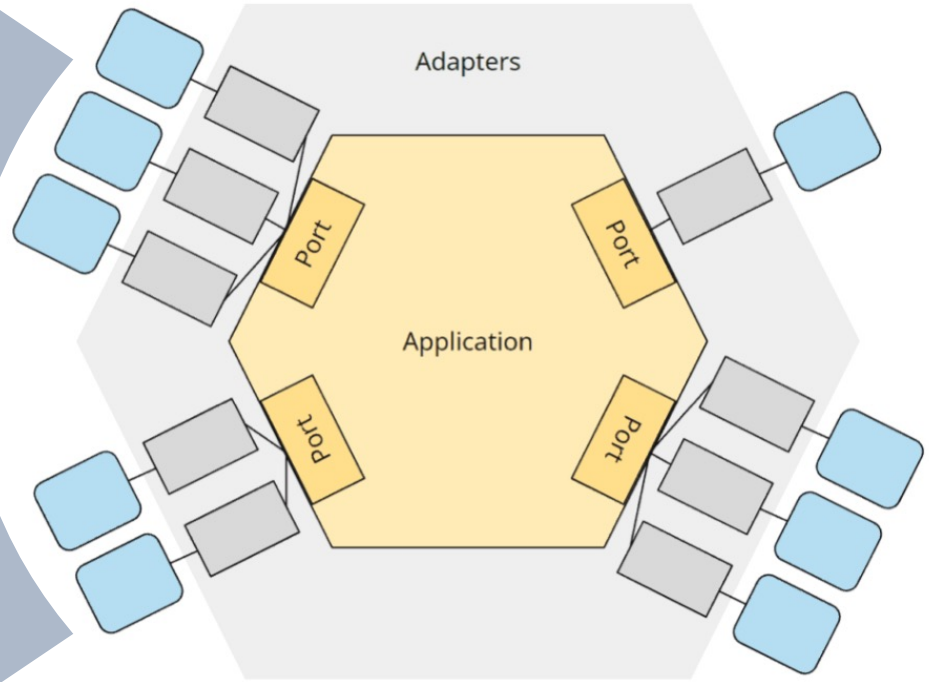


Adapters

Application

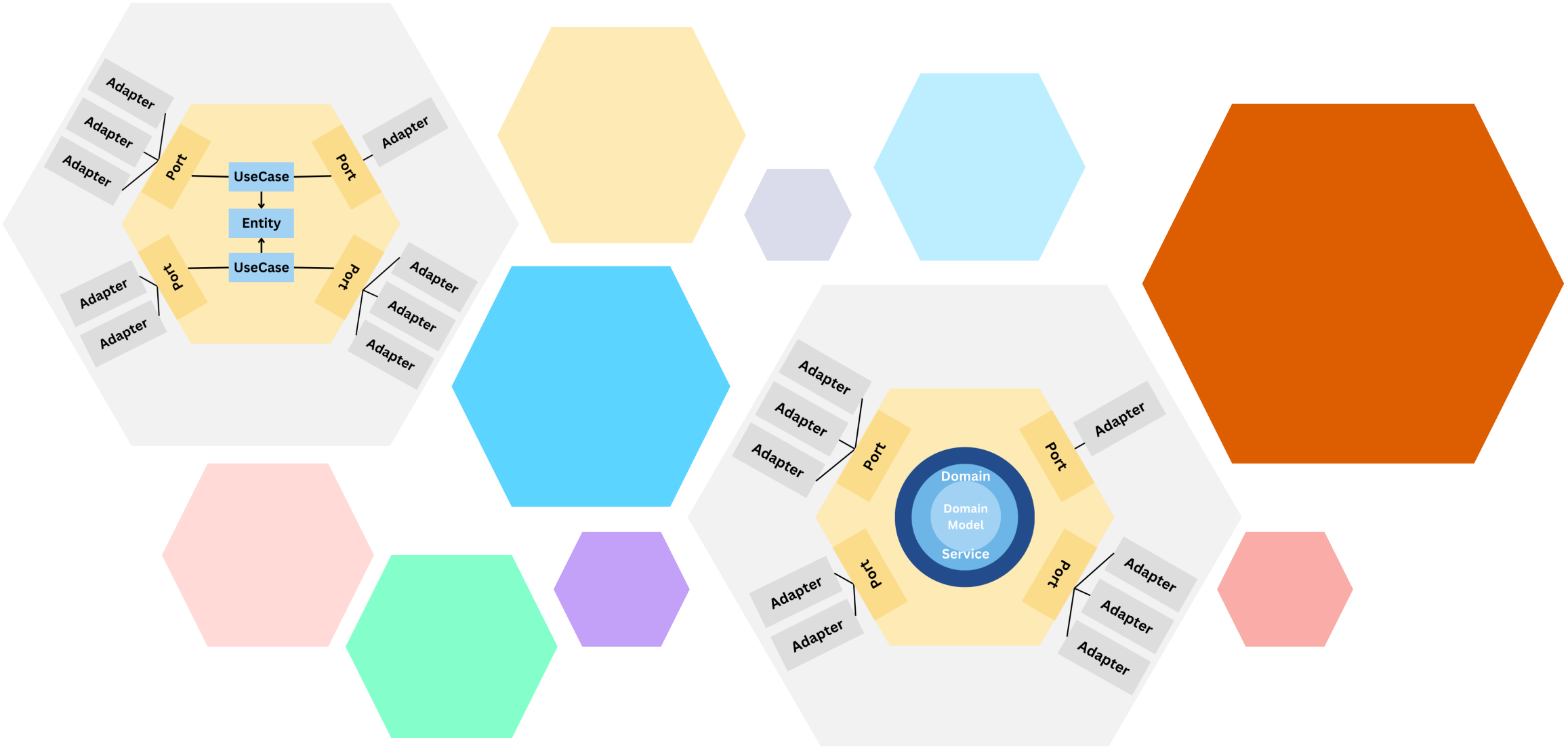


„What do you see inside the Application?“



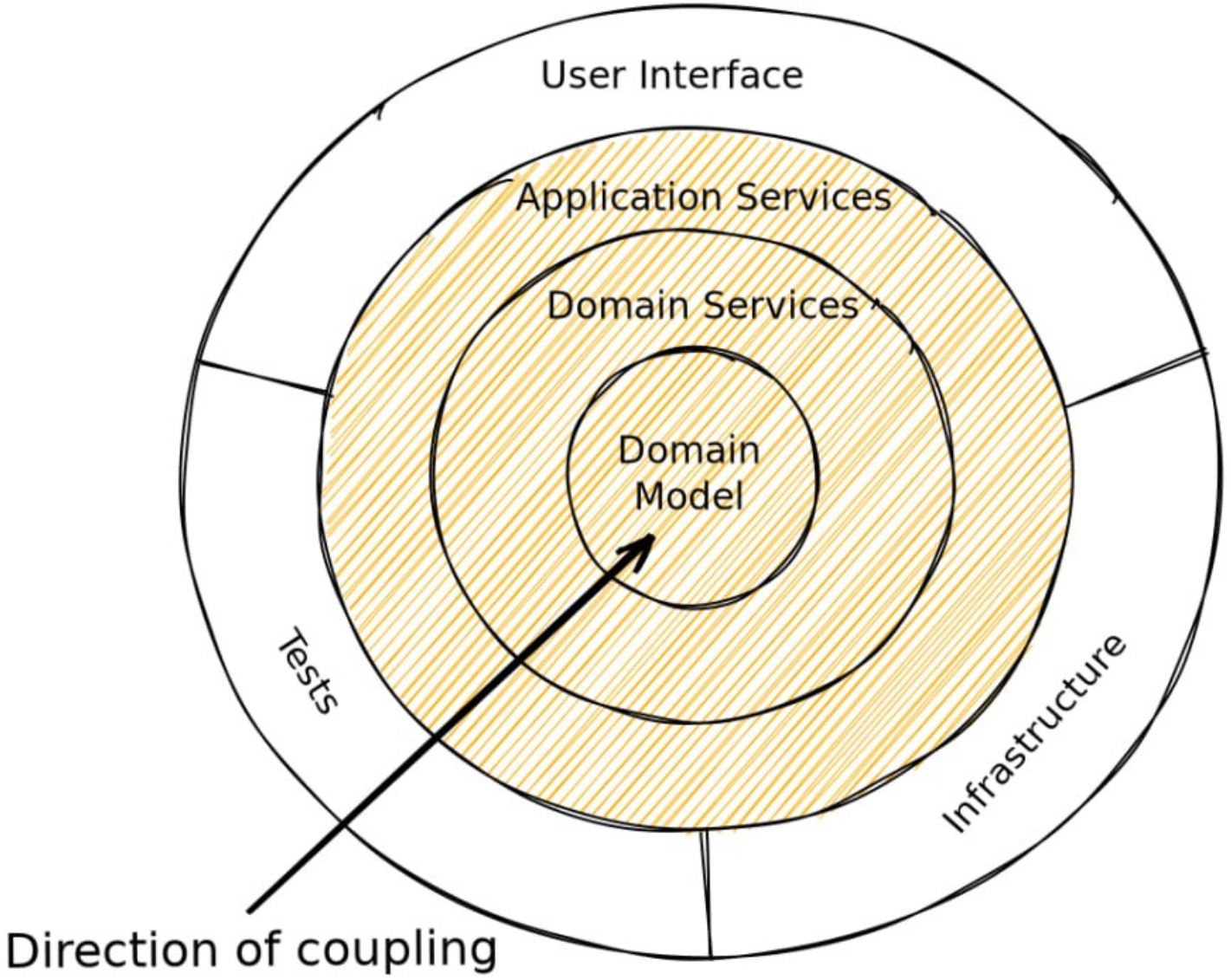
I don’t care – not my business

ALISTAIR COCKBURN

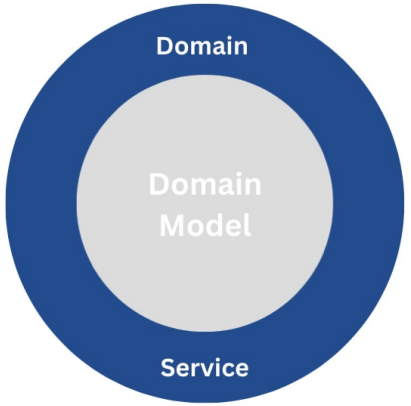

Adapter
Adapter
Adapter
Port
UseCase
Entity
UseCase
Port
Adapter
Port
Adapter
Adapter
Adapter
Port
Adapter
Adapter
Adapter
Adapter
Adapter
Port
Domain
Domain Model
Service
Port
Adapter
Adapter
Adapter
Port
Adapter
Adapter
Adapter
Port
Adapter
Adapter
Adapter

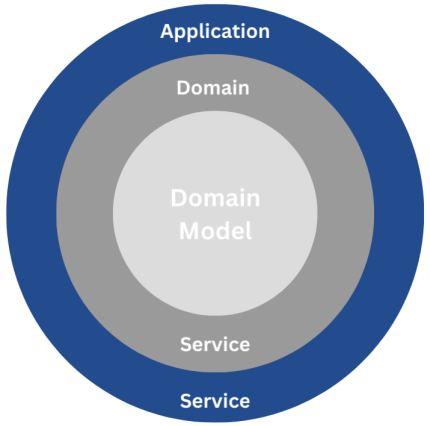
Onion Architektur
Jeffrey Palermo

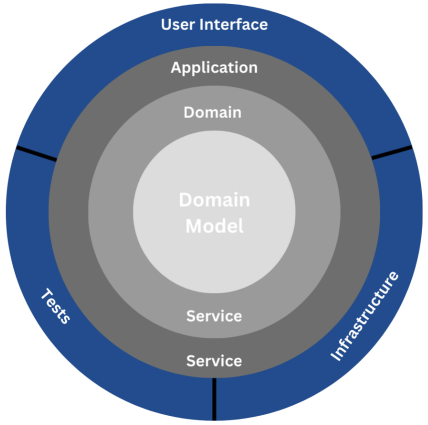
Onion Architektur

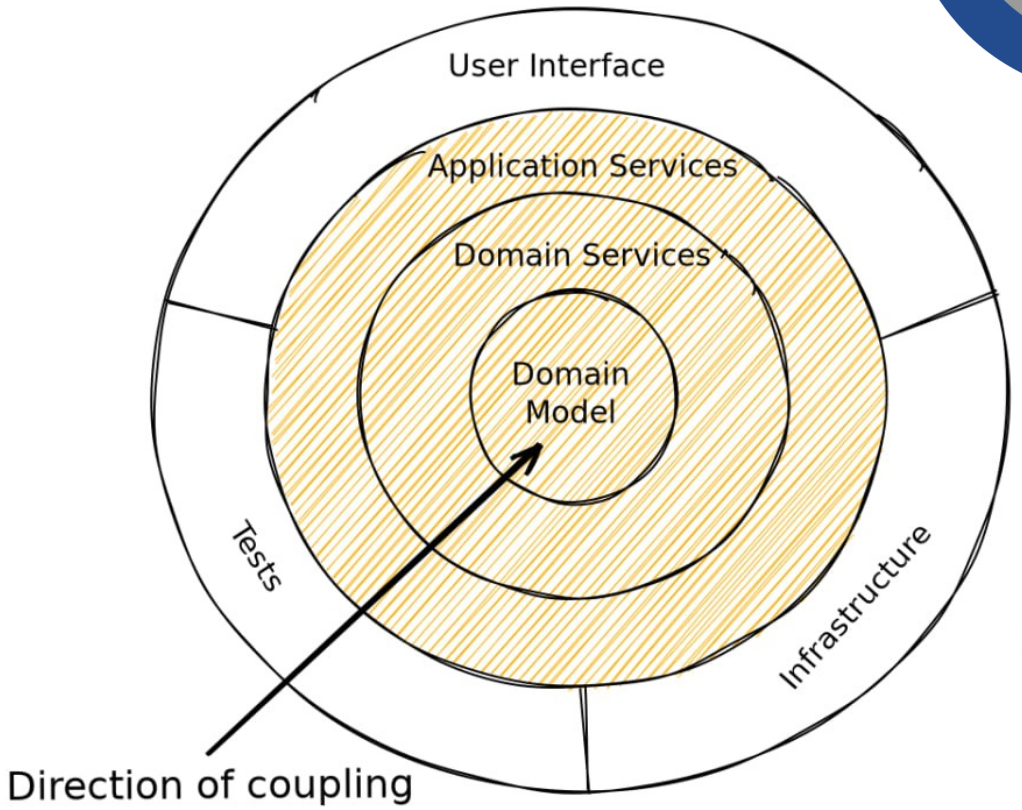


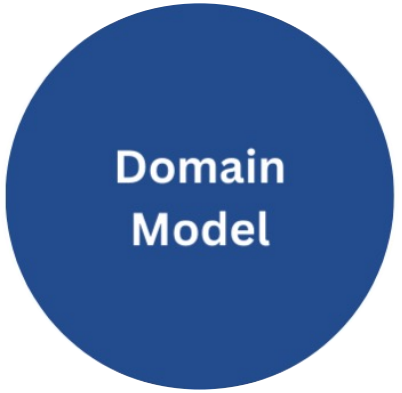
Onion Architektur





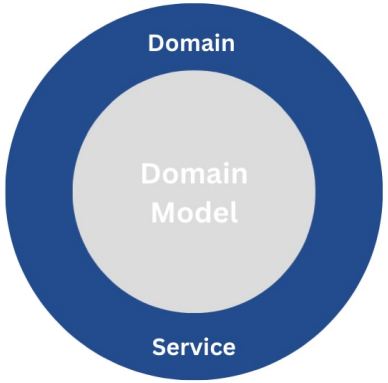


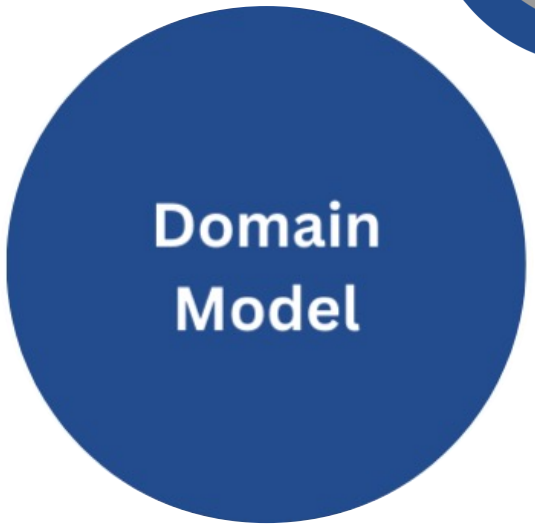


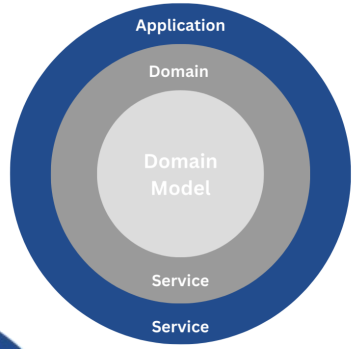


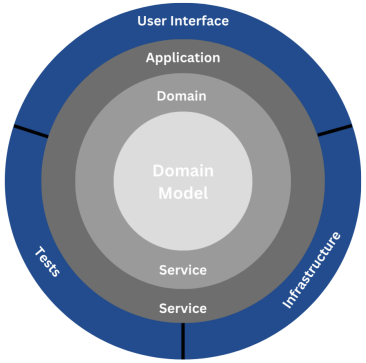
Onion Architektur – Domain Model

- Essenzielle Geschäftslogik und Geschäftsregeln
- Kombination von Zustand und Verhalten
- Die Grundlage für die gesamte Softwarearchitektur

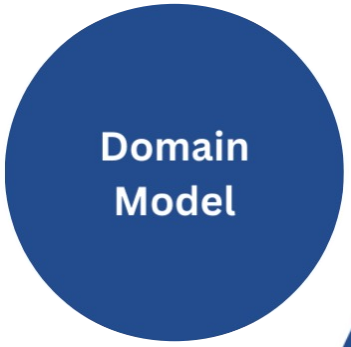




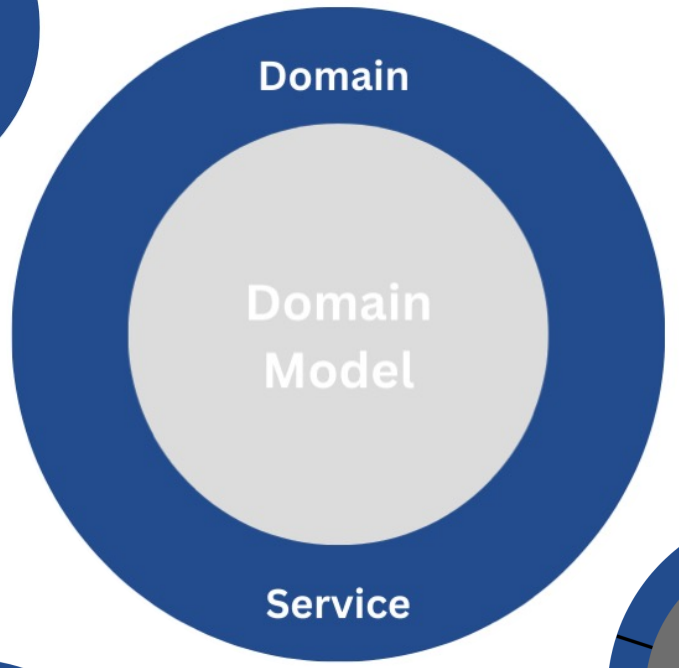


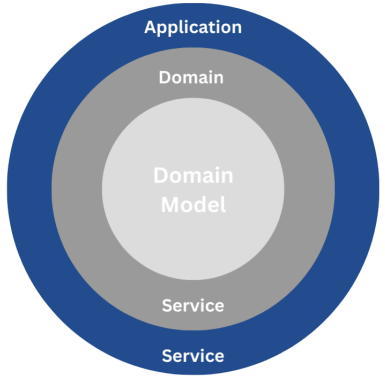


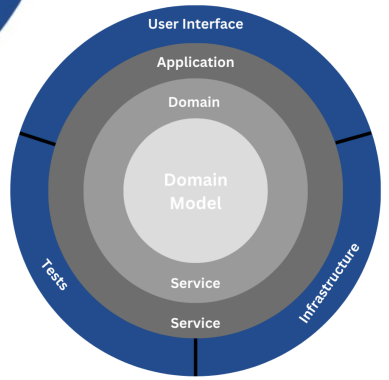
Onion Architektur – Domain Service



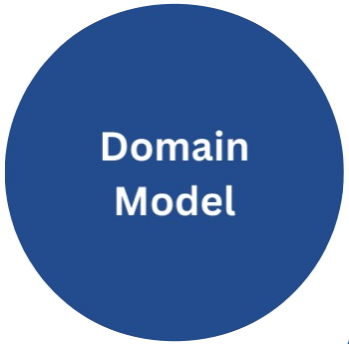
- Repräsentation von Verhalten keine Identität
- Zustandslos
- Werden oft verwendet, um Änderungen an mehreren Aggregates und Entities zu koordinieren.

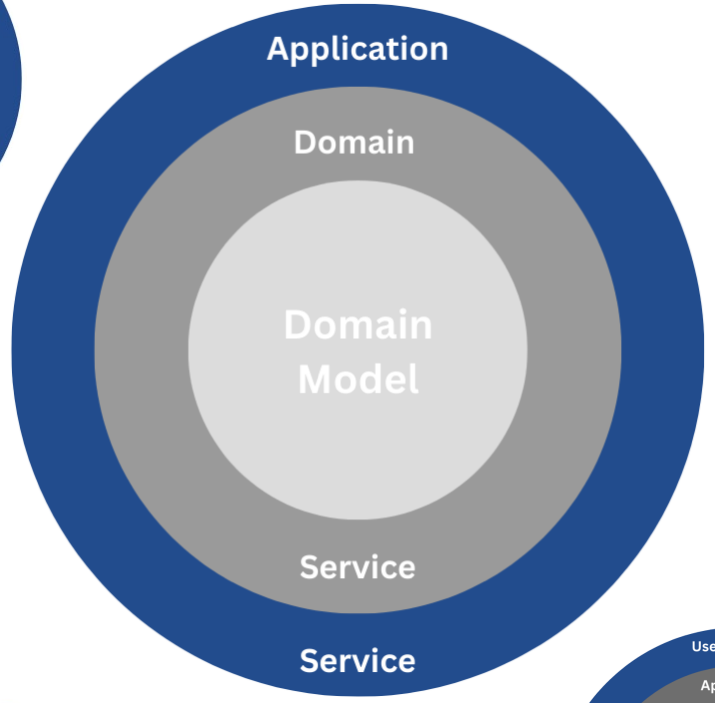




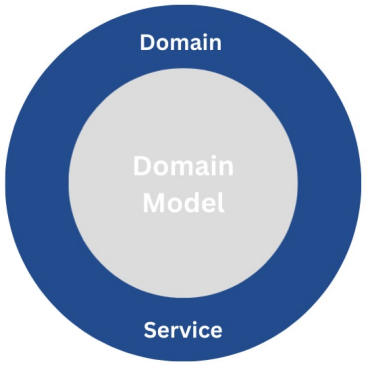


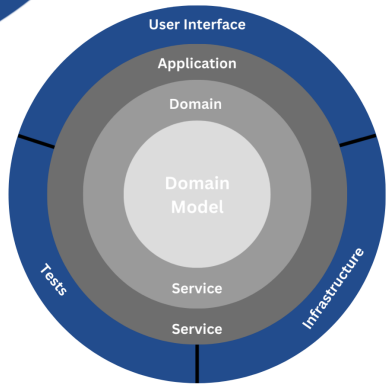
Onion Architektur – Application Service





- Die Schicht des Application Services repräsentiert die Anwendungsfälle und das Verhalten der Anwendung
- Enthält Anwendungslogik zur Koordination der Anwendungsfälle
- Delegiert an Domänen- und Infrastrukturschichten
- Verbirgt Details der Domänenschicht

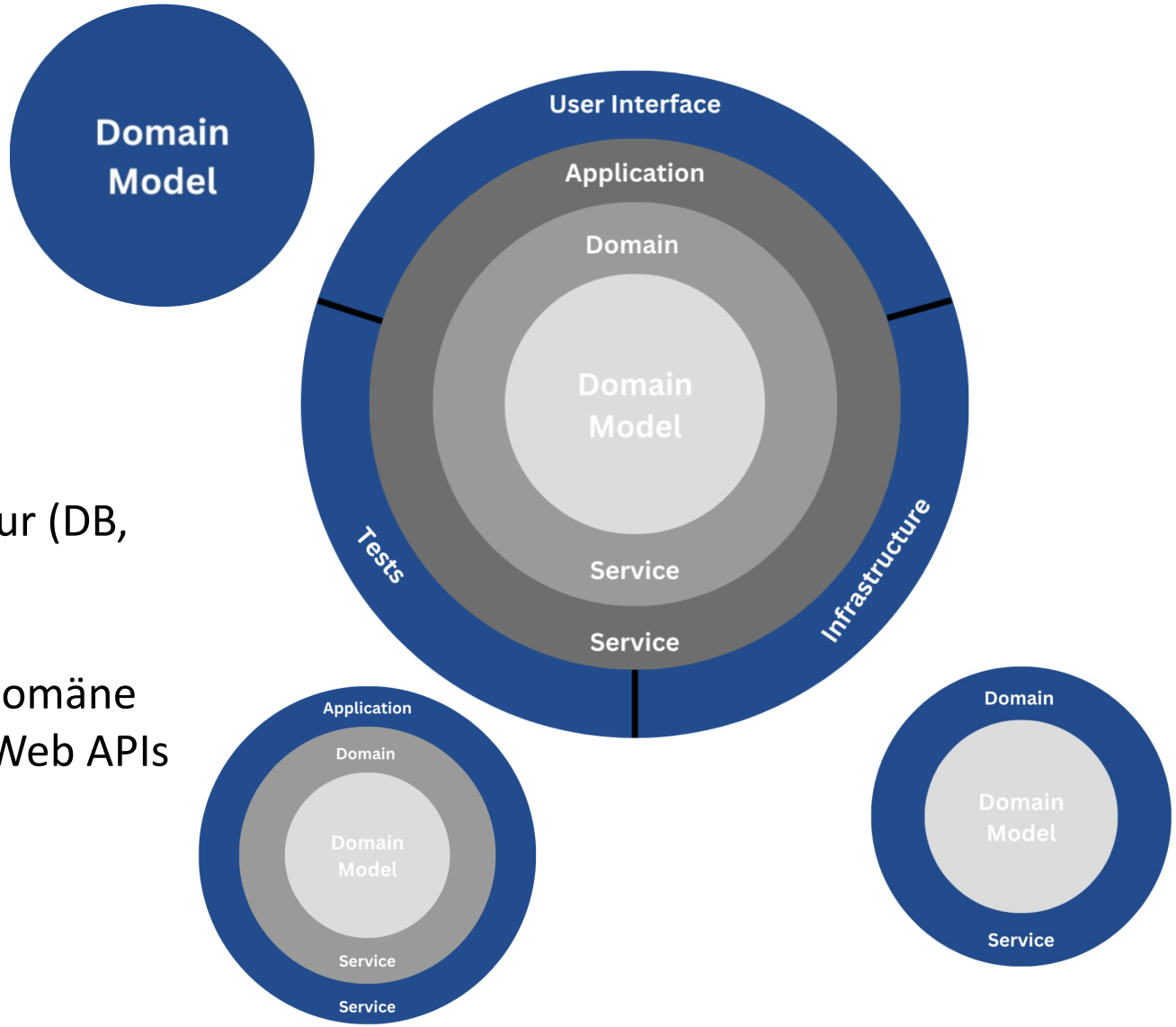




```java
@Service
public class ApplicationService {
    //...
    public void legeArtikelInDenWarenkorb(UUID artikelId, int anzahl, UUID warenkorbID) {
        Warenkorb warenkorb = getWarenkorb(warenkorbID);
        Artikel artikel = getArtikel(artikelId);
        Lager lager = lagerRepository.findeMit(artikelId);

        domainService
            .legeArtikelInDenWarenkorb(warenkorb, artikel, new Anzahl(anzahl), lager);

        warenkorbRepository.speichere(warenkorb);
        lagerRepository.speichere(lager);
    }
    //...
}
```

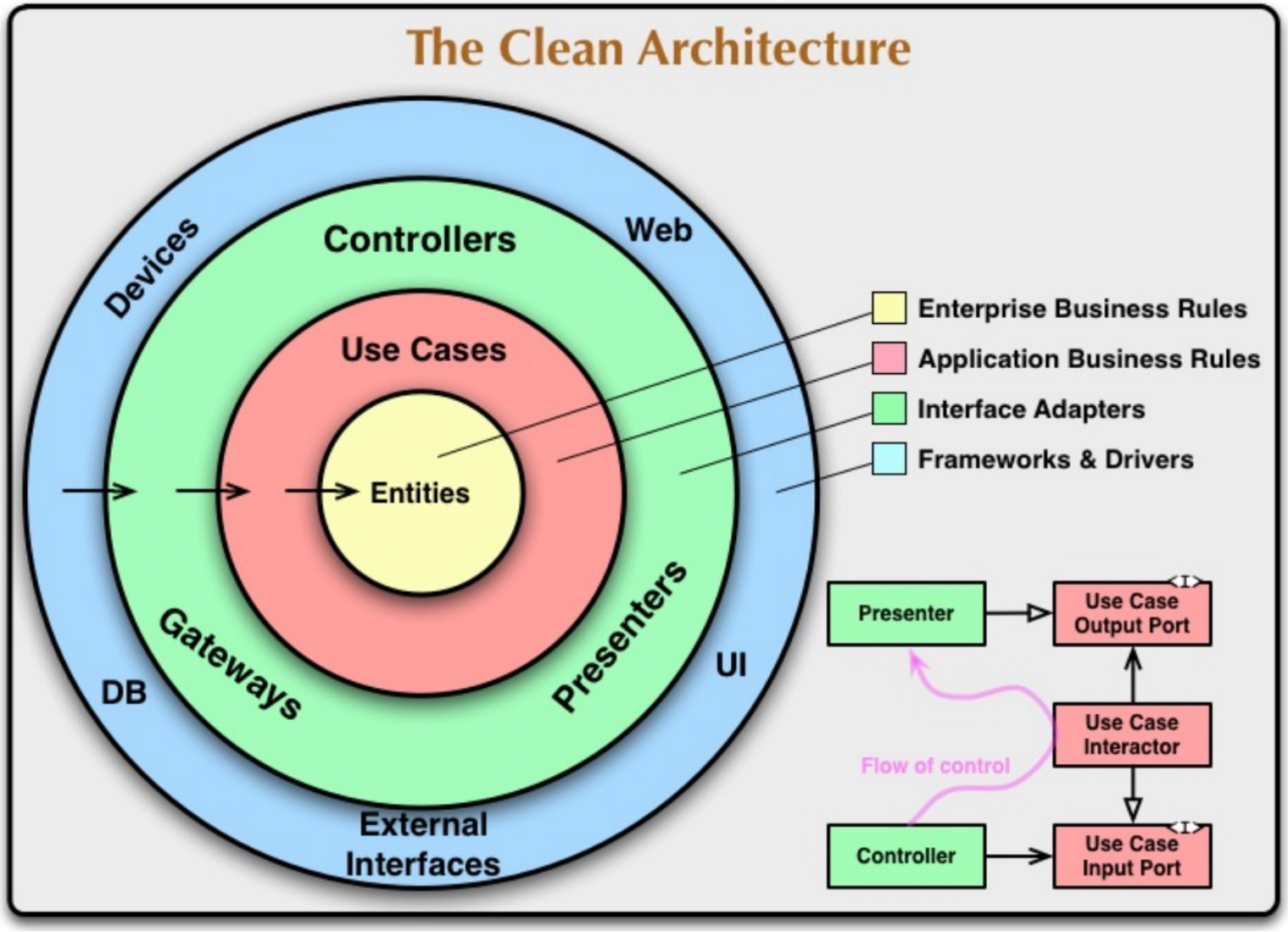
Onion Architektur – Infrastruktur

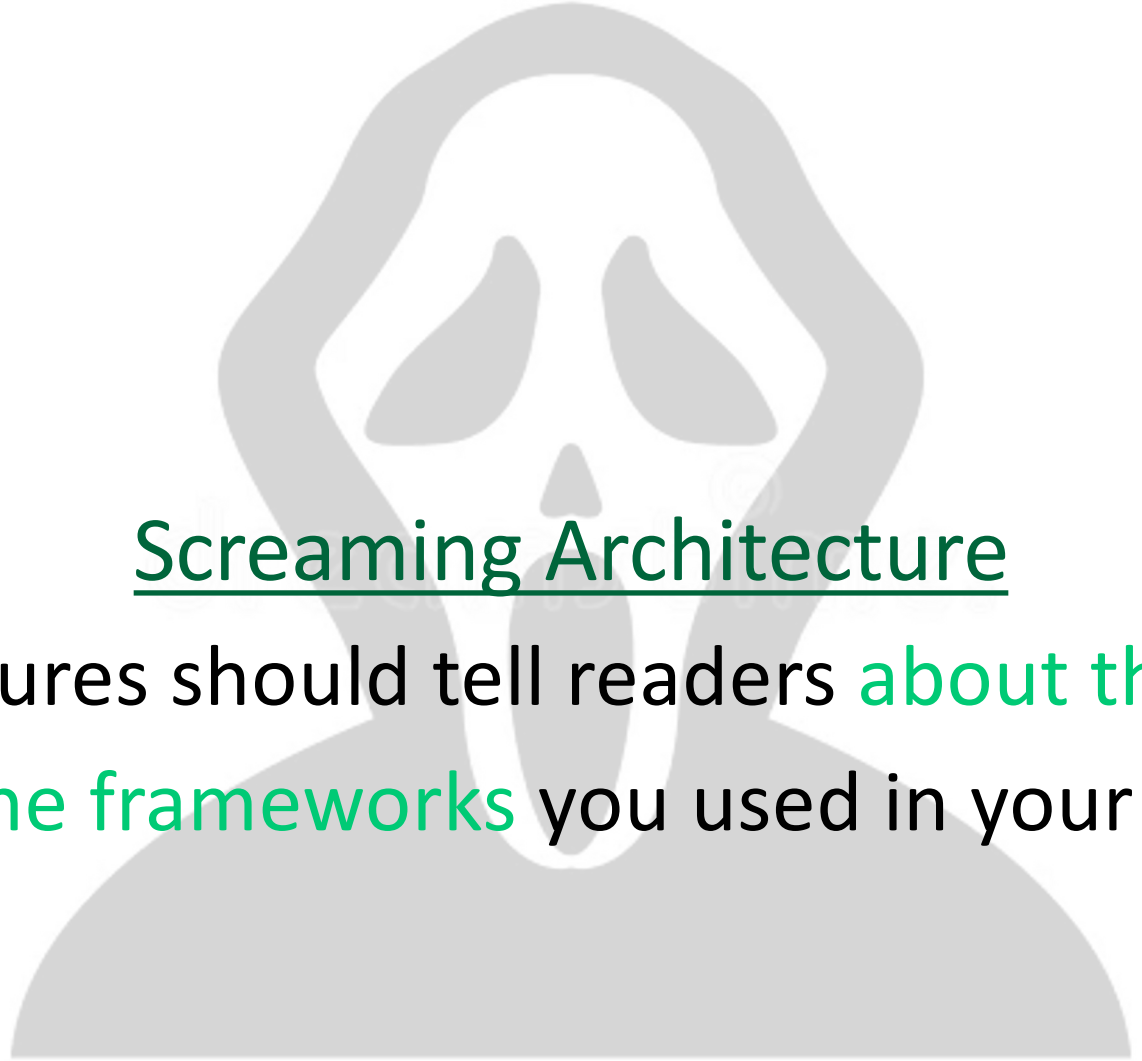
- Äußerste Schicht der Onion Architektur
- Konzentriert sich auf technische Aspekte der Anwendung
- Enthält:
  - Frameworks zur Anbindung an Infrastruktur (DB, Message Bus, Umsysteme, etc.)
  - User Interface/Web APIs
  - Fachliche Tests gegen Funktionalität der Domäne
  - Tests für den Application Service und die Web APIs



Clean Architektur
Robert C. Martin

Clean Architektur



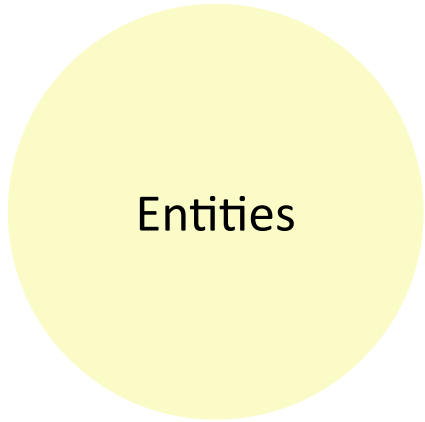


Screaming Architecture

Your architectures should tell readers about the system, not about the frameworks you used in your system.

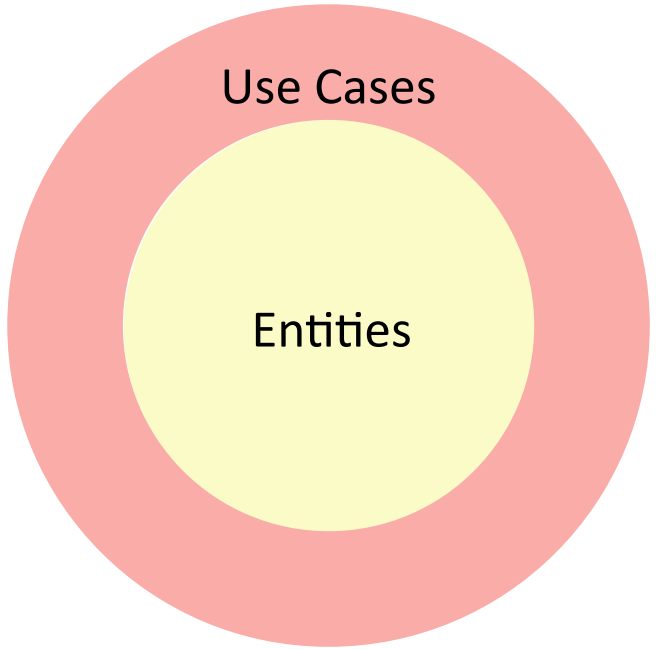
Clean Architektur – Entities

- Innere Schicht
- Unternehmensweite kritische Geschäftsregeln
- Umsetzung als Objekt mit Methoden oder als eine Reihe von Datenstrukturen und Funktionen
- Bildung der grundlegenden Geschäftsobjekte



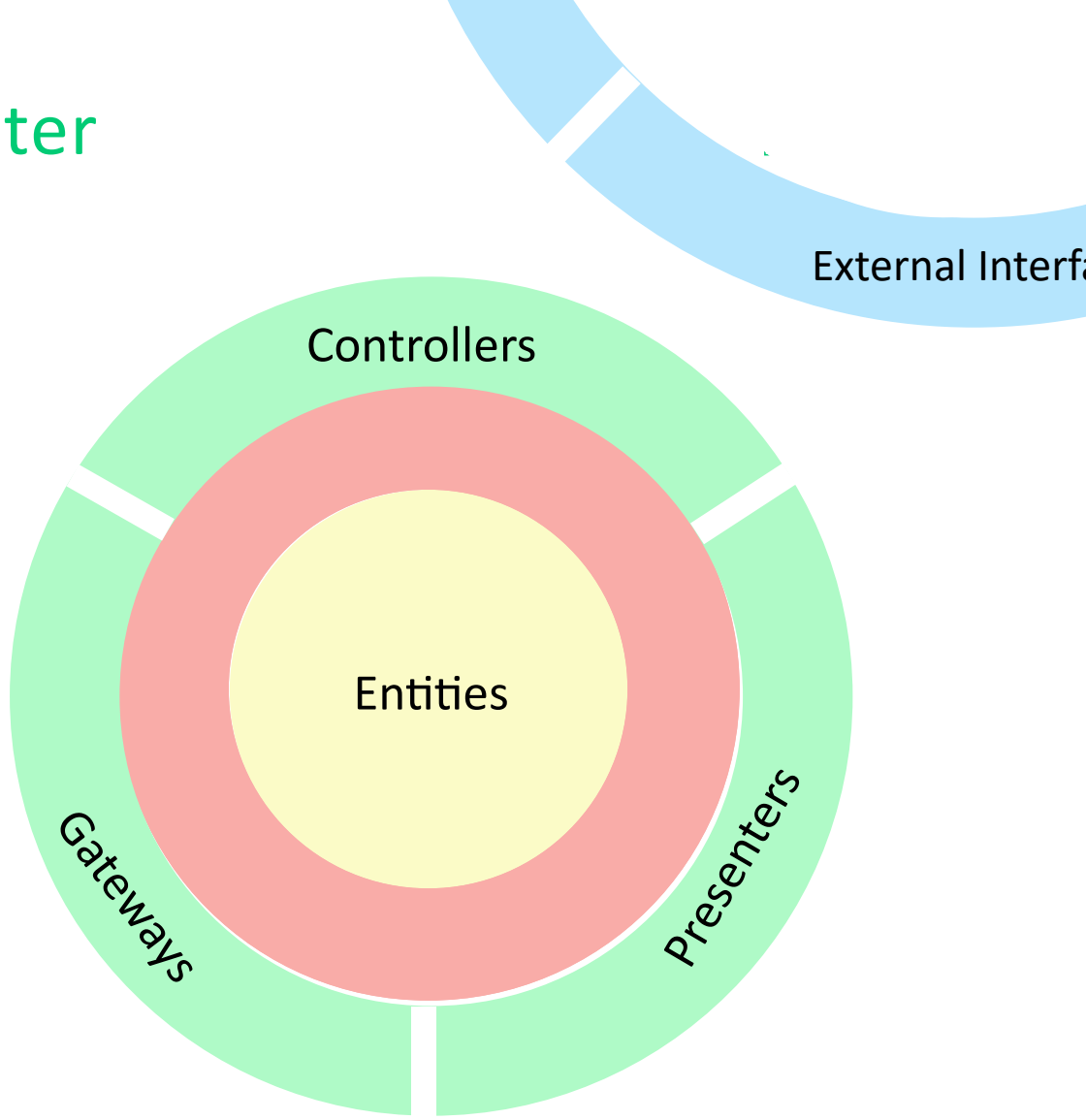
Clean Architektur – Use Cases

- Die Schicht der Anwendungsfälle
- Anwendungsspezifische Geschäftslogik
- Implementierung Anwendungsfälle des Systems
- Orchestrierung den Aufrufen von den Entities



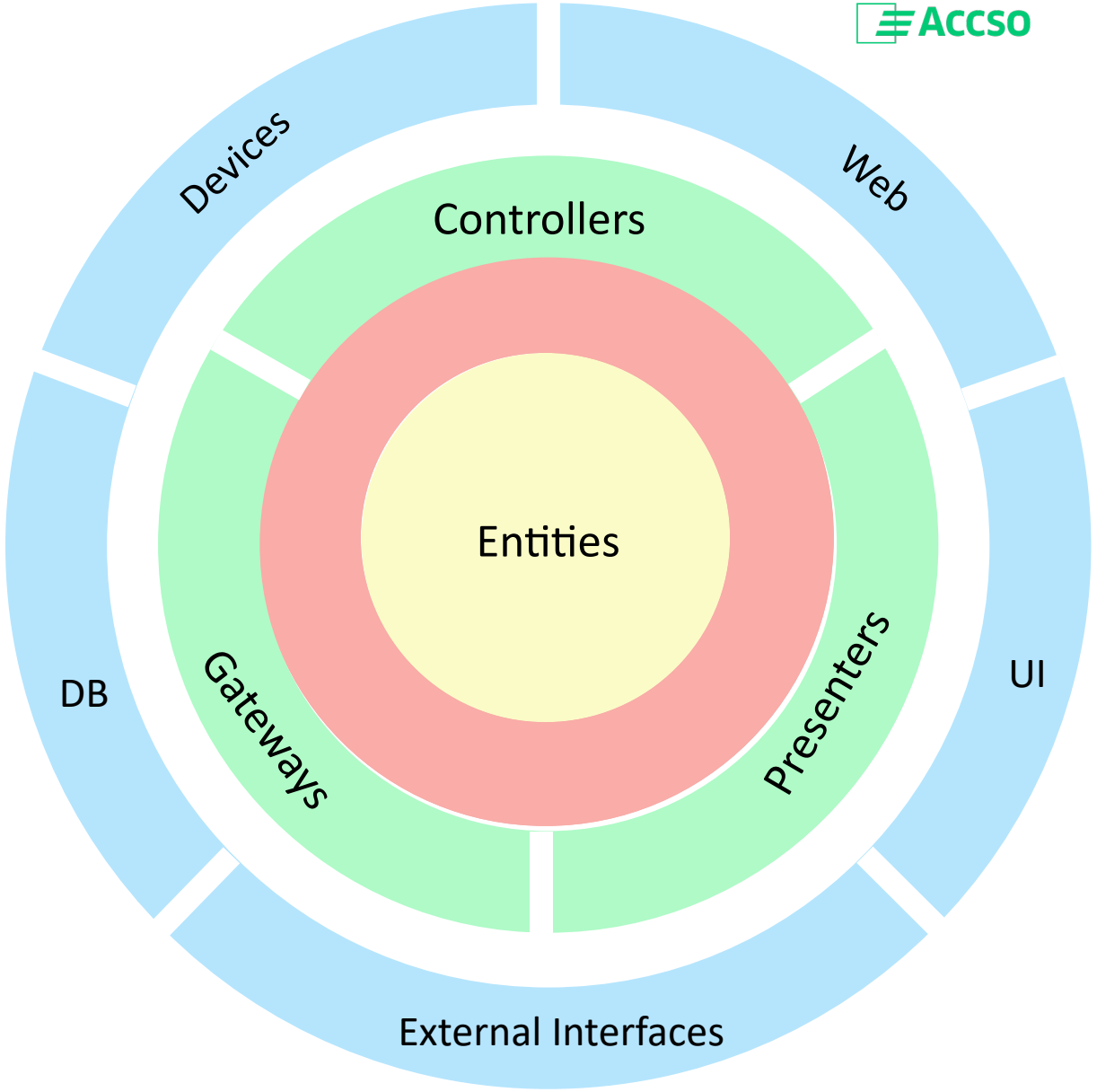
Clean Architektur – Schnittstellenadapter

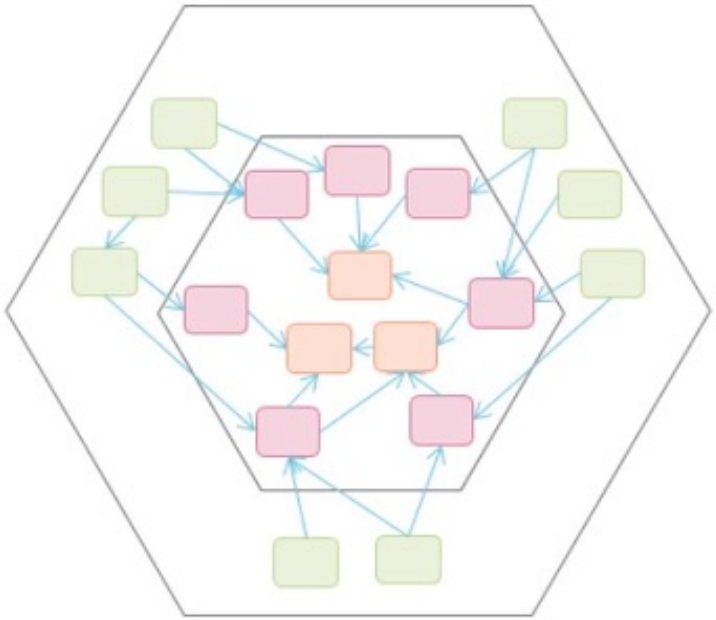
- Die Schicht der Schnittstellenadapter
- Adapter für Datenformatanpassung
- Anpassung an externe Komponenten:
  - Datenbank und UI
  - Die Presenter, Views und Controller einer MVC-Architektur



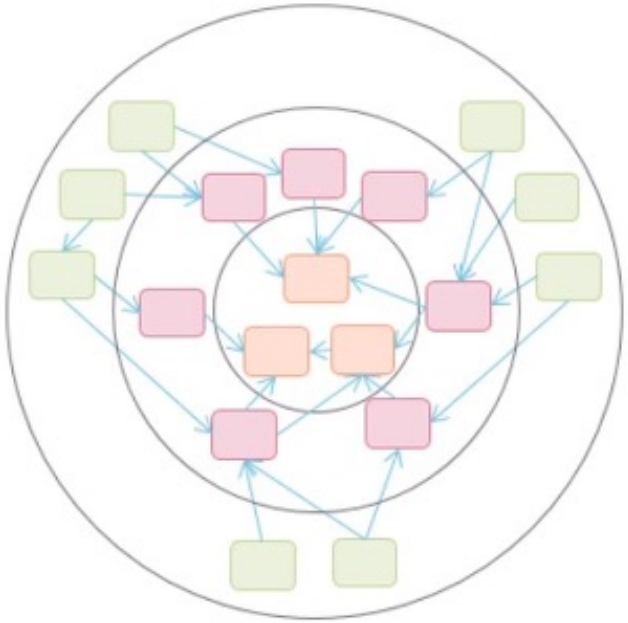
Clean Architektur – Frameworks

- Die Schicht der Frameworks
- Technische Details für externe Systemintegration
- Datenbank und Web-Frameworks

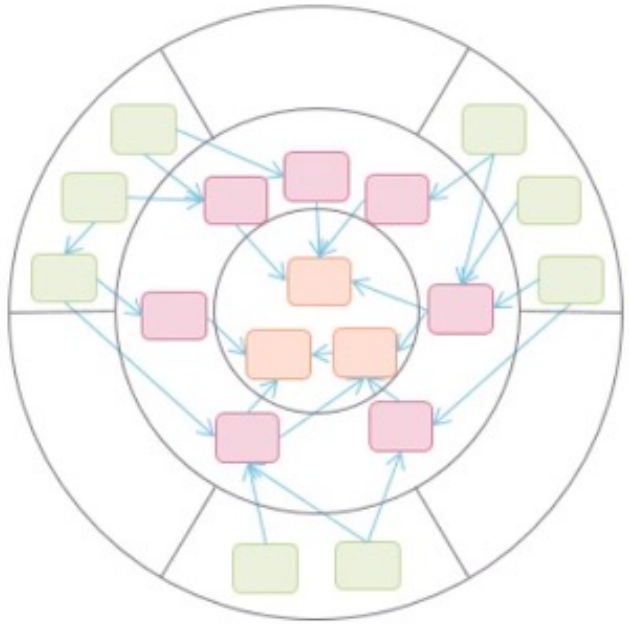




Hexagonal

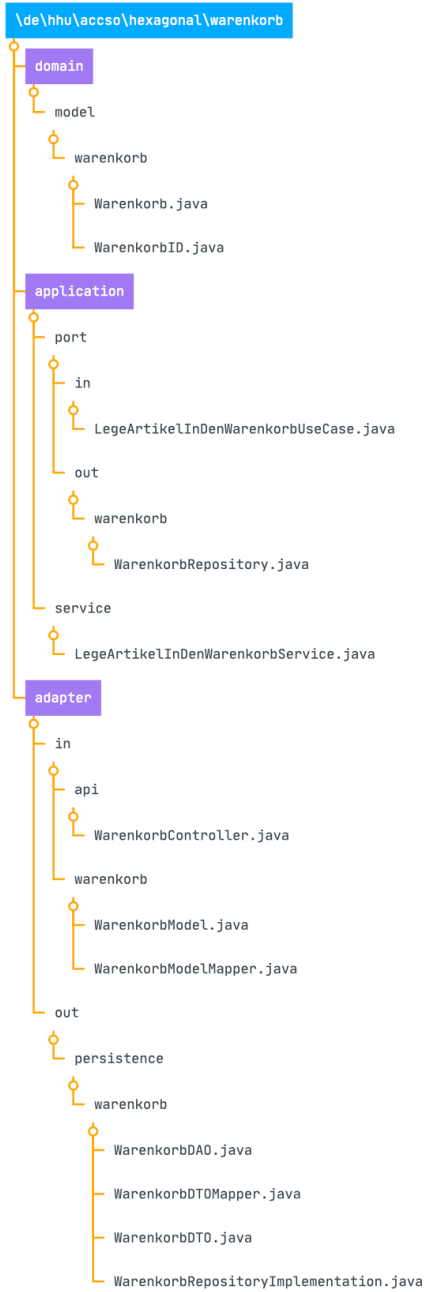


Onion

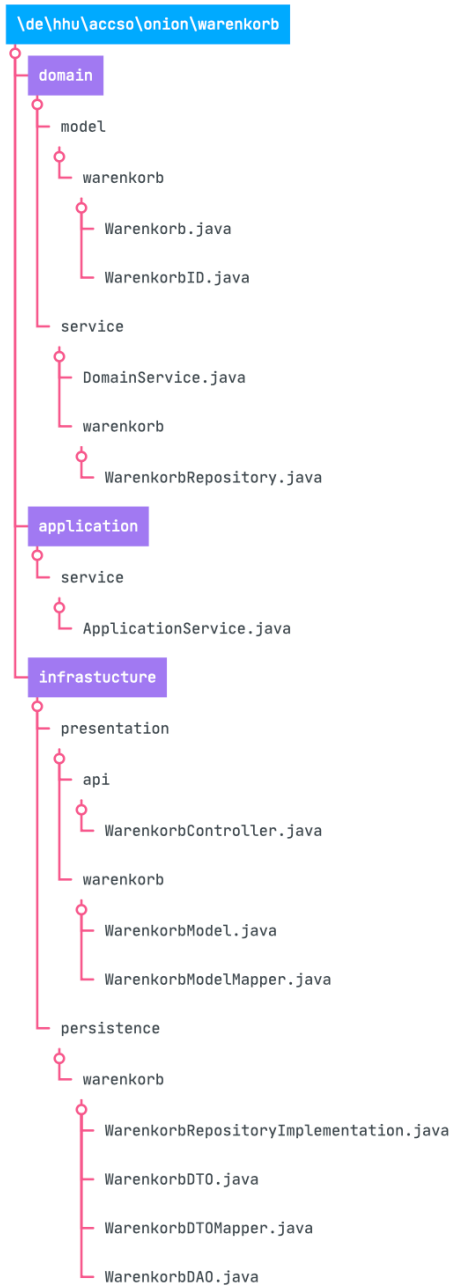


Clean

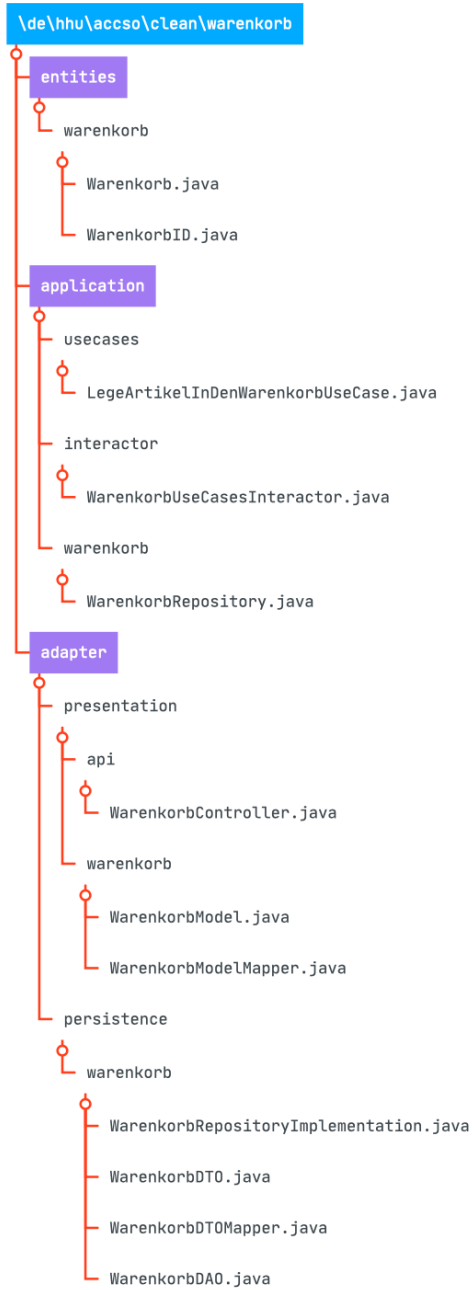
Hexagonale Architektur



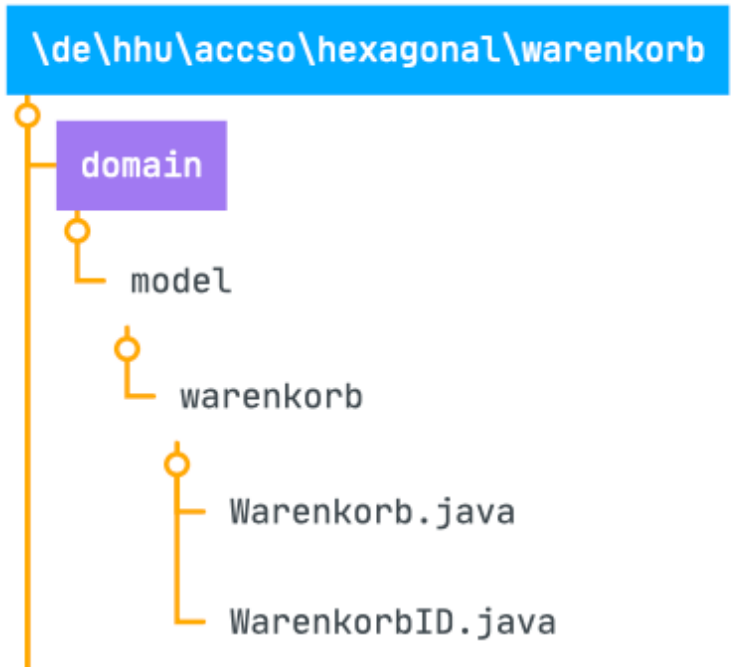
Onion Architektur



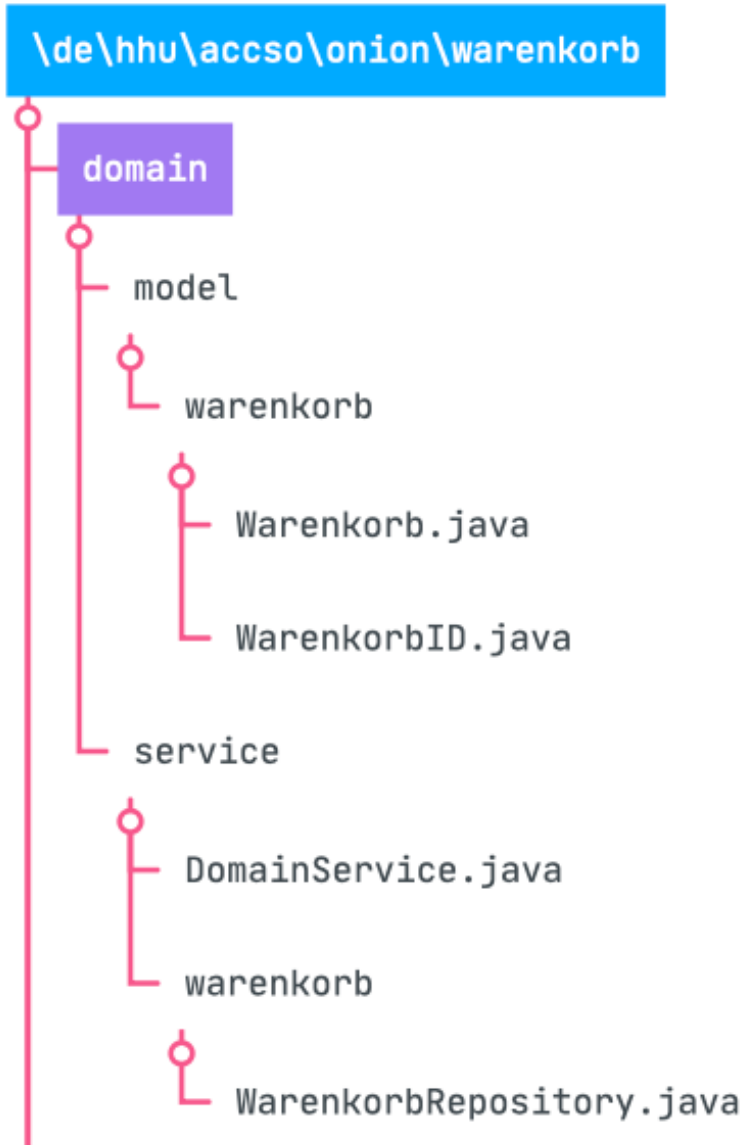
Clean Architektur



Hexagonale Architektur



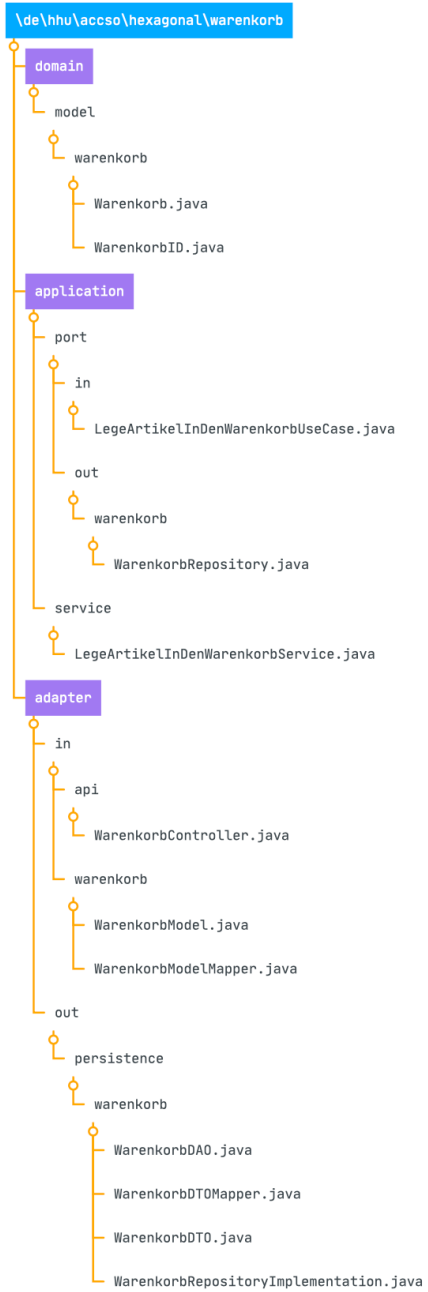
Onion Architektur



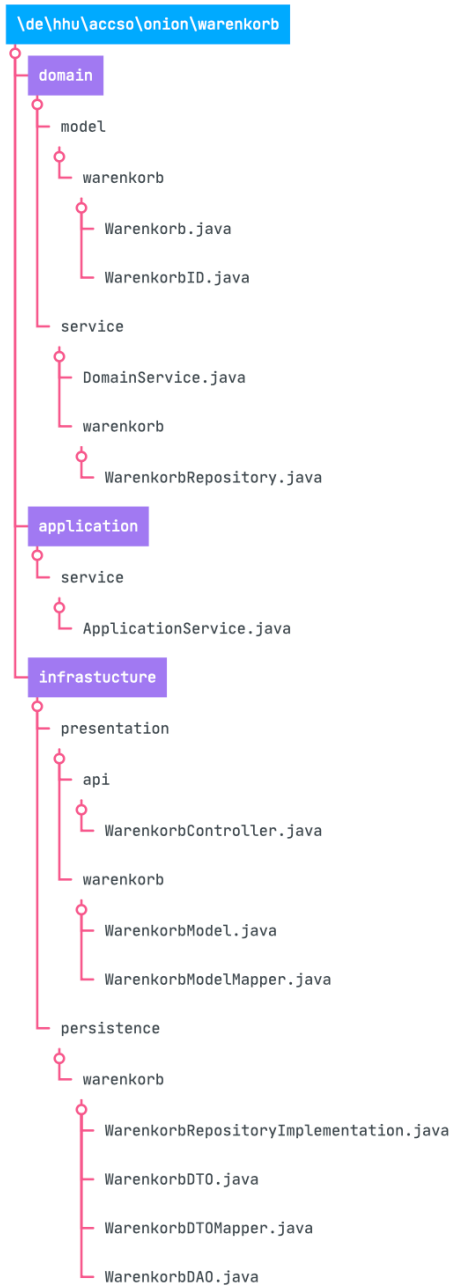
Clean Architektur

Accso

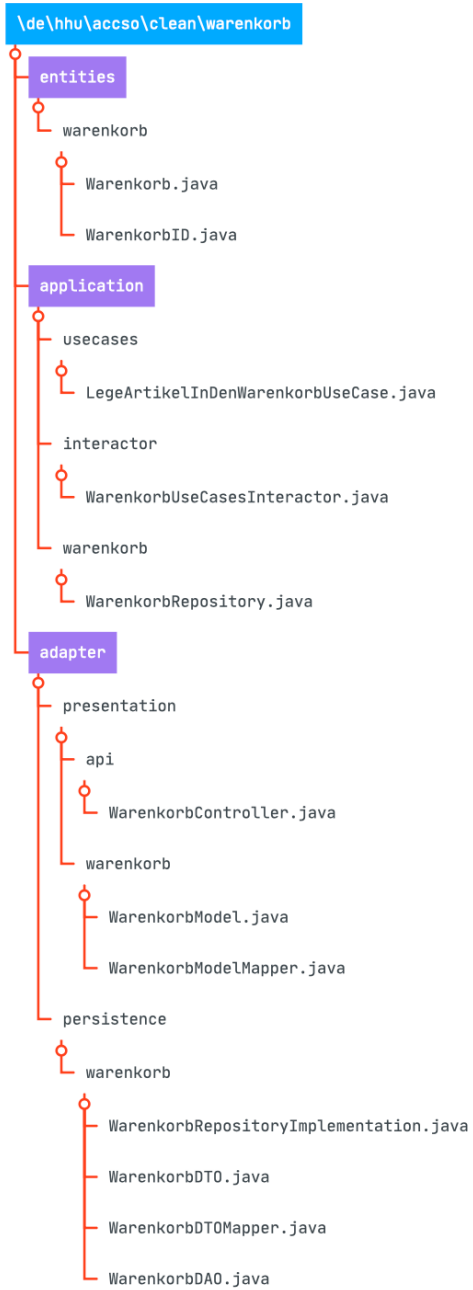
Hexagonale Architektur



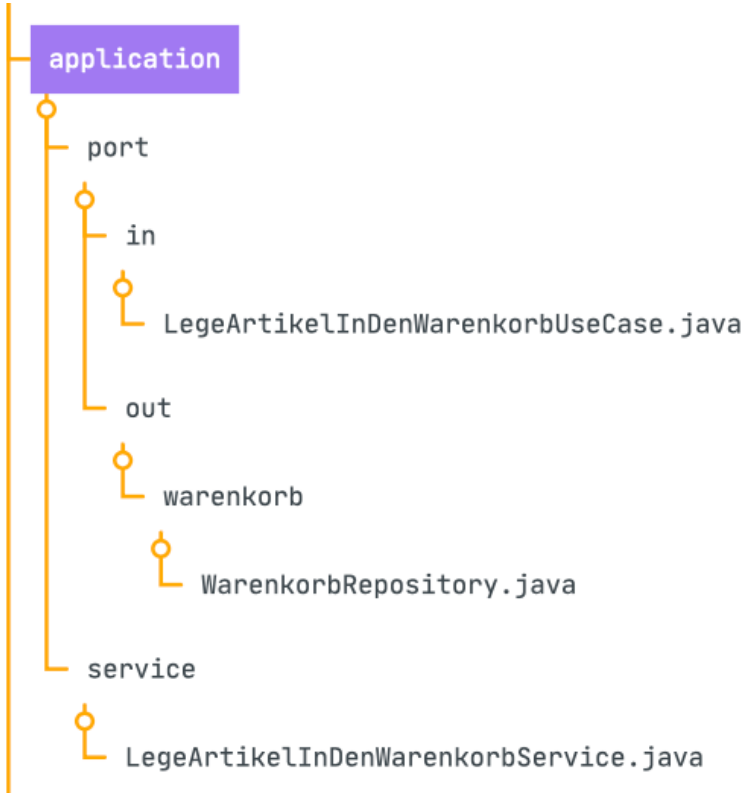
Onion Architektur



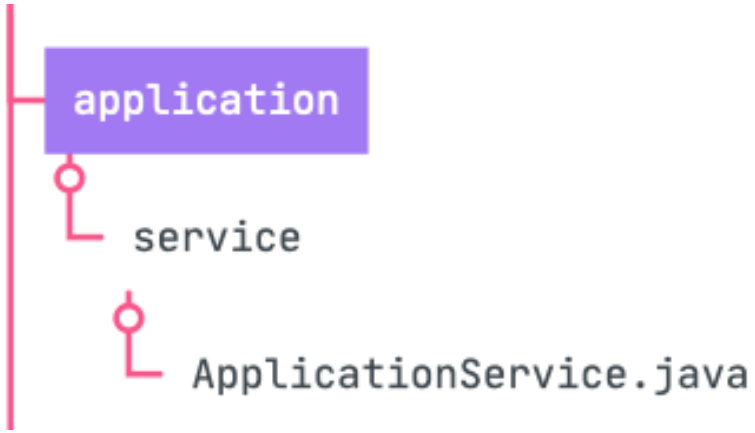
Clean Architektur



Hexagonale Architektur

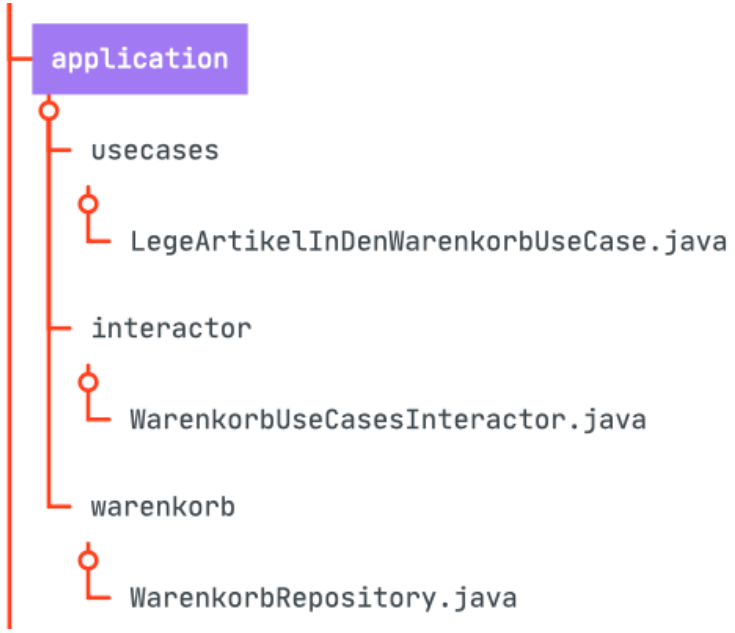


Onion Architektur

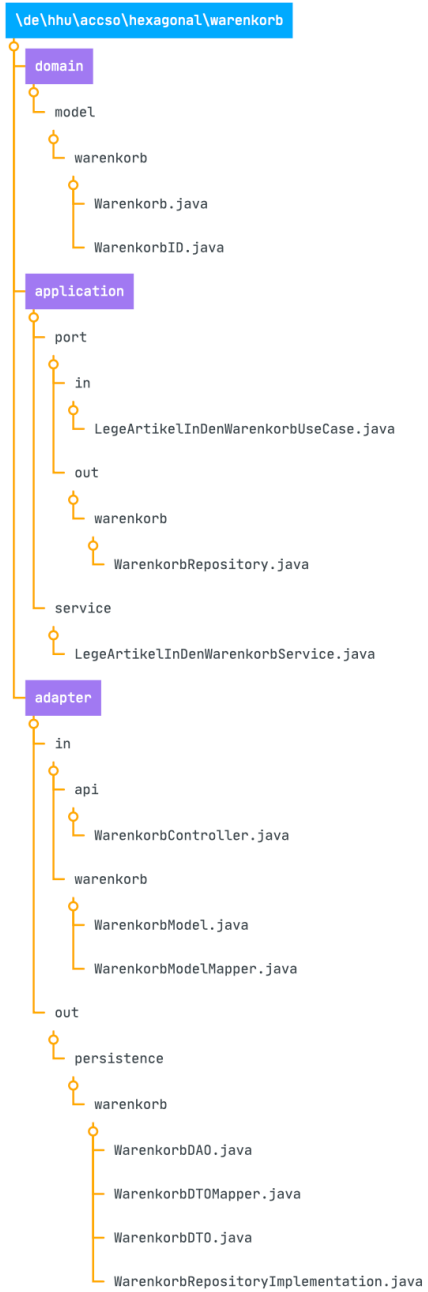


Clean Architektur

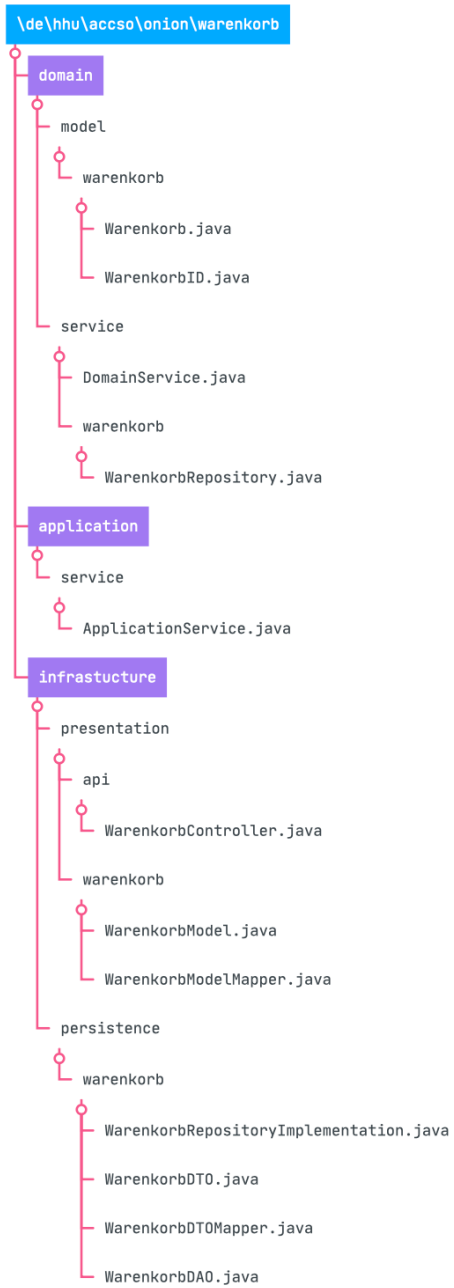
Accso



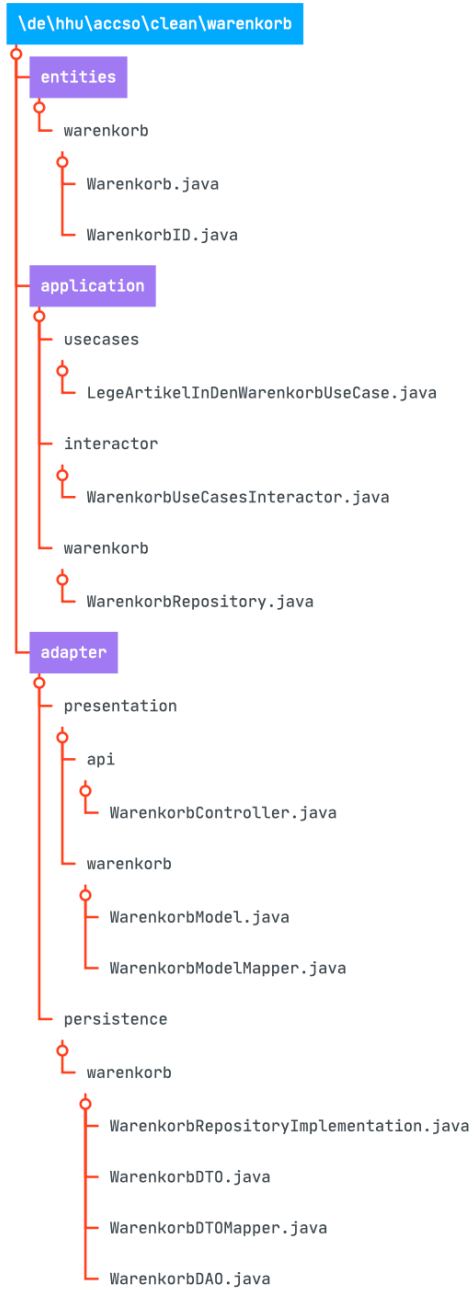
Hexagonale Architektur



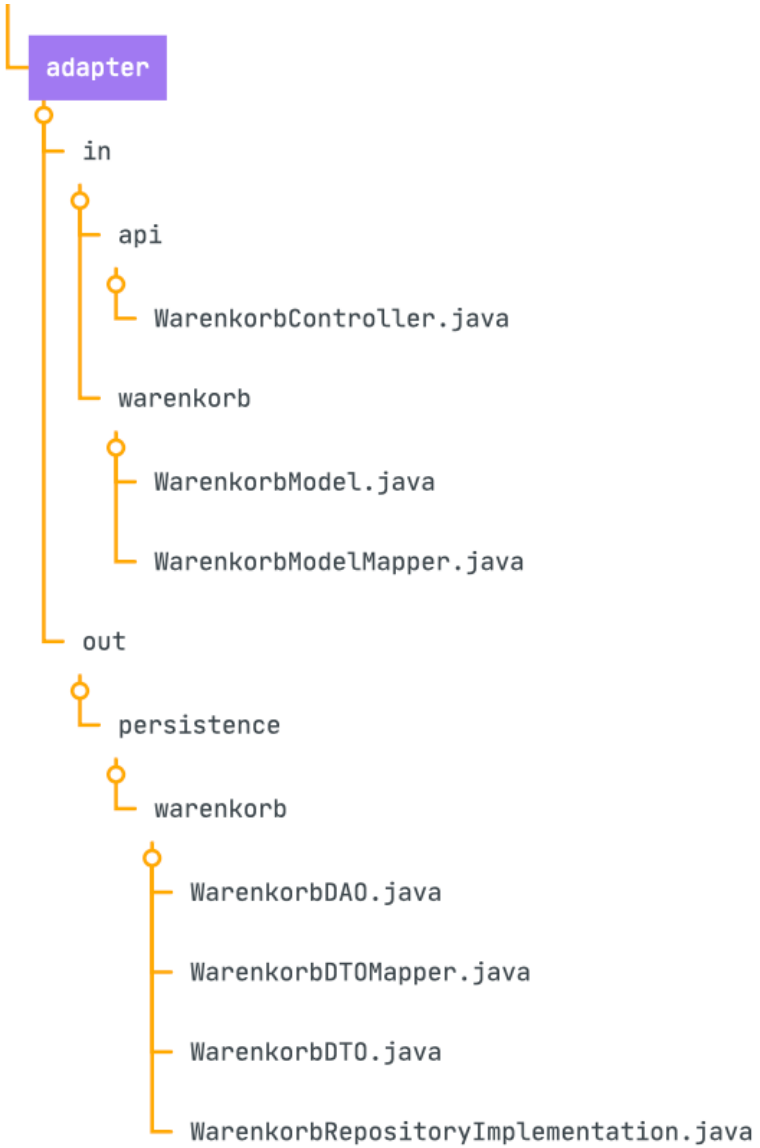
Onion Architektur



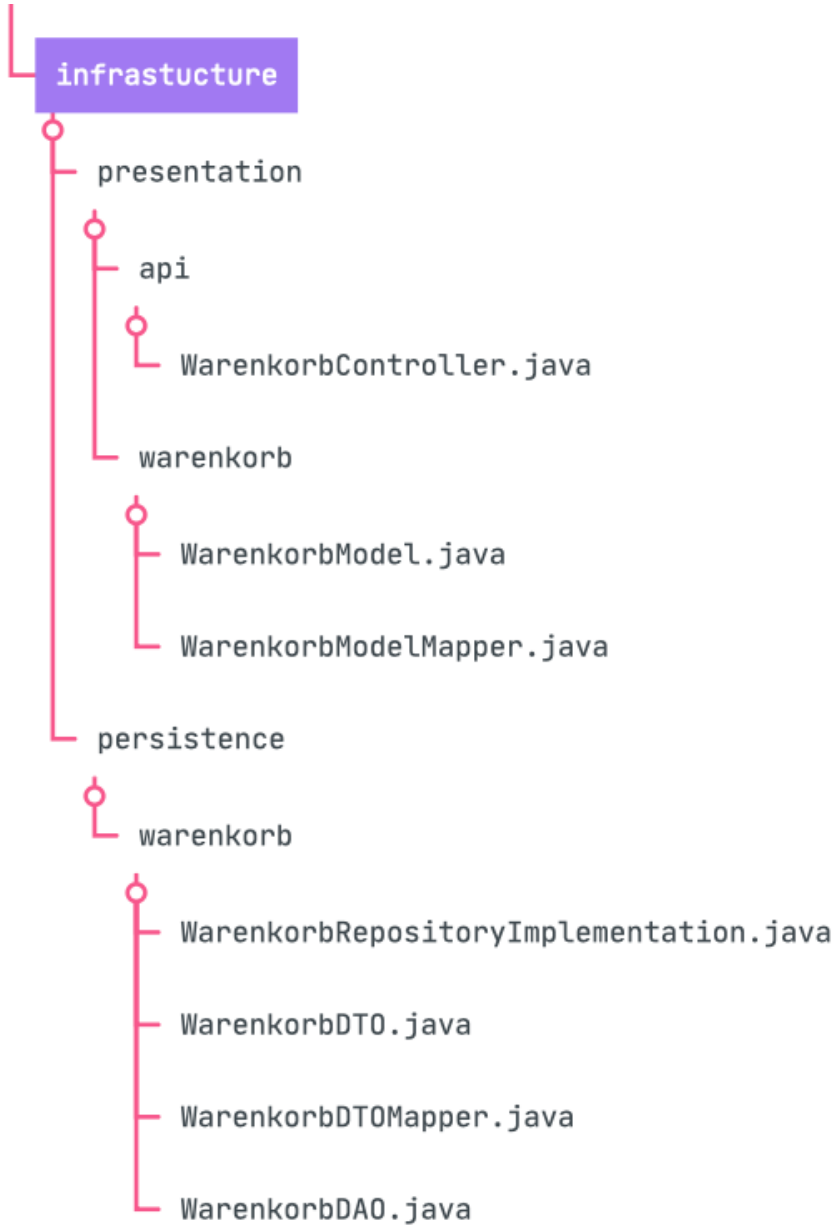
Clean Architektur



Hexagonale Architektur

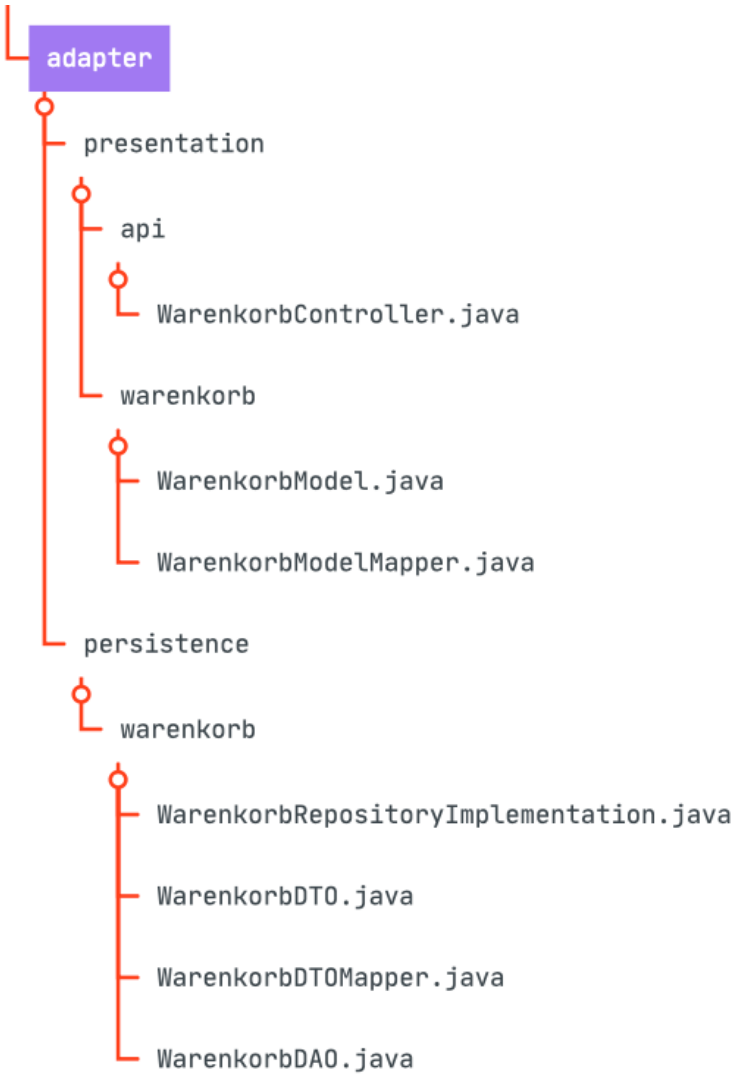


Onion Architektur

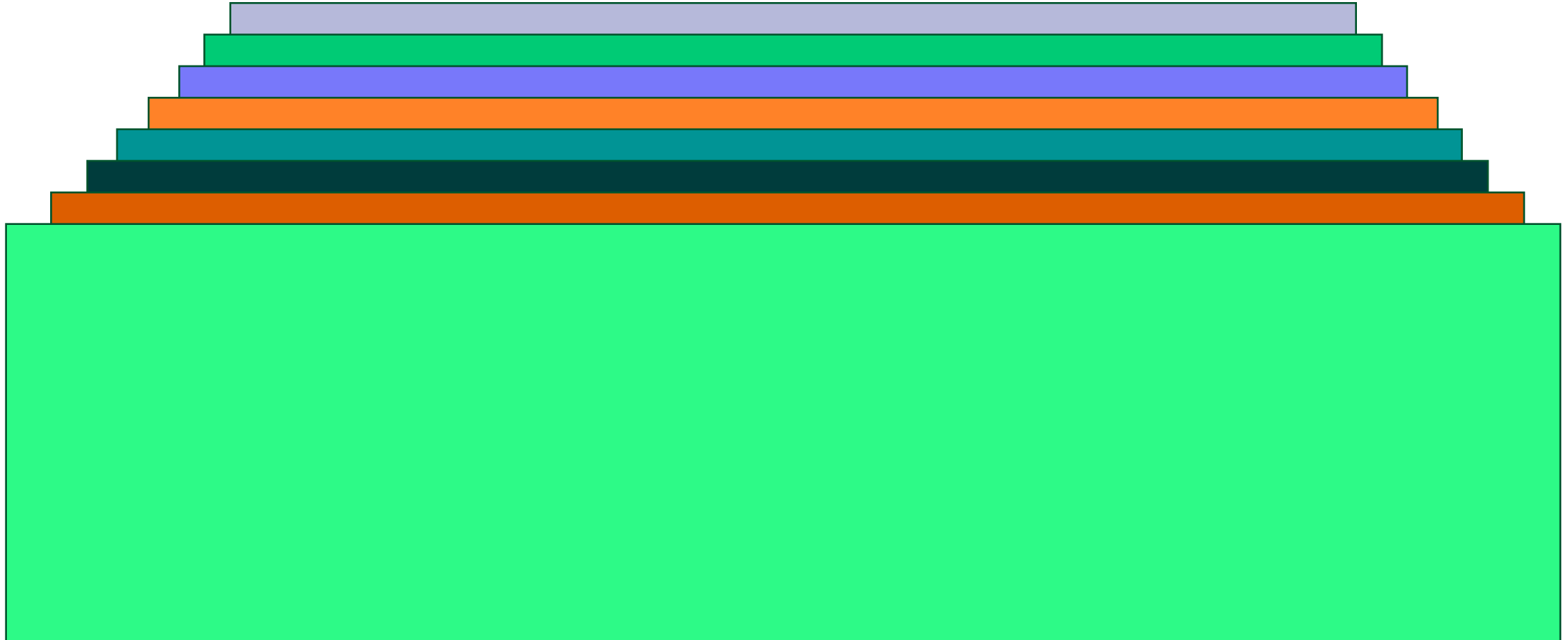


Clean Architektur

Accso



Gemeinsamkeiten



Gemeinsamkeit #1

Zentralisierte Geschäftslogik

Gemeinsamkeit #2

Dependency Rule

Accso

Gemeinsamkeit #3

Geschäftslogik ist unabhängig von Infrastruktur, Datenbank, UI und Frameworks

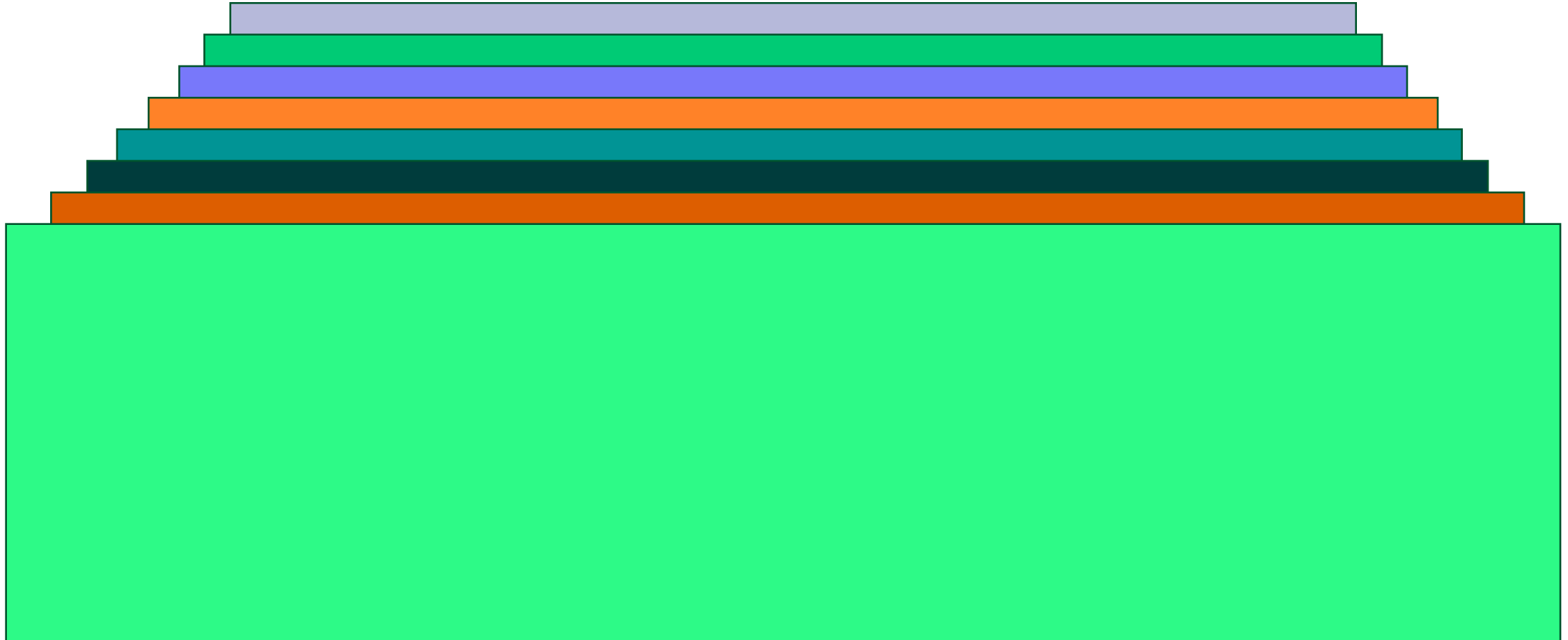
Gemeinsamkeit #4

Accso

Mapping

Accso

Unterschiede



Unterschied #1

Die Konkretheit der Architekturdefinition

Unterschied #2

Accso

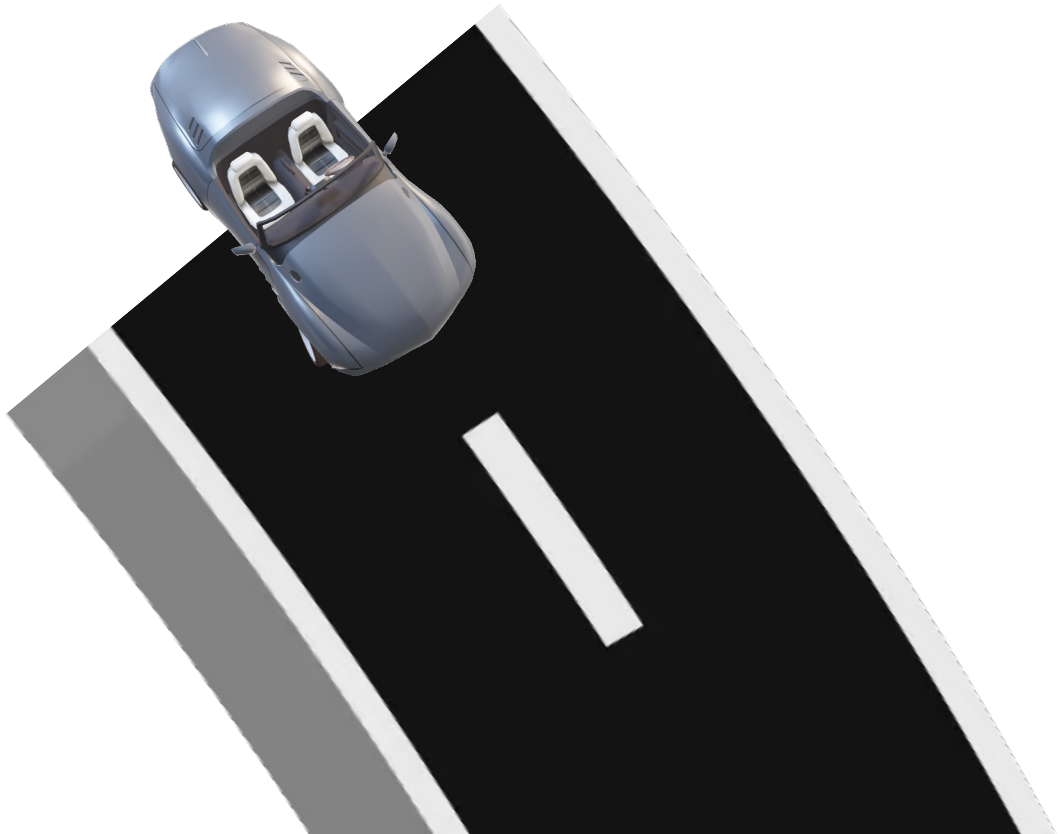
Die Abhängigkeit – Regel

Unterschied #3

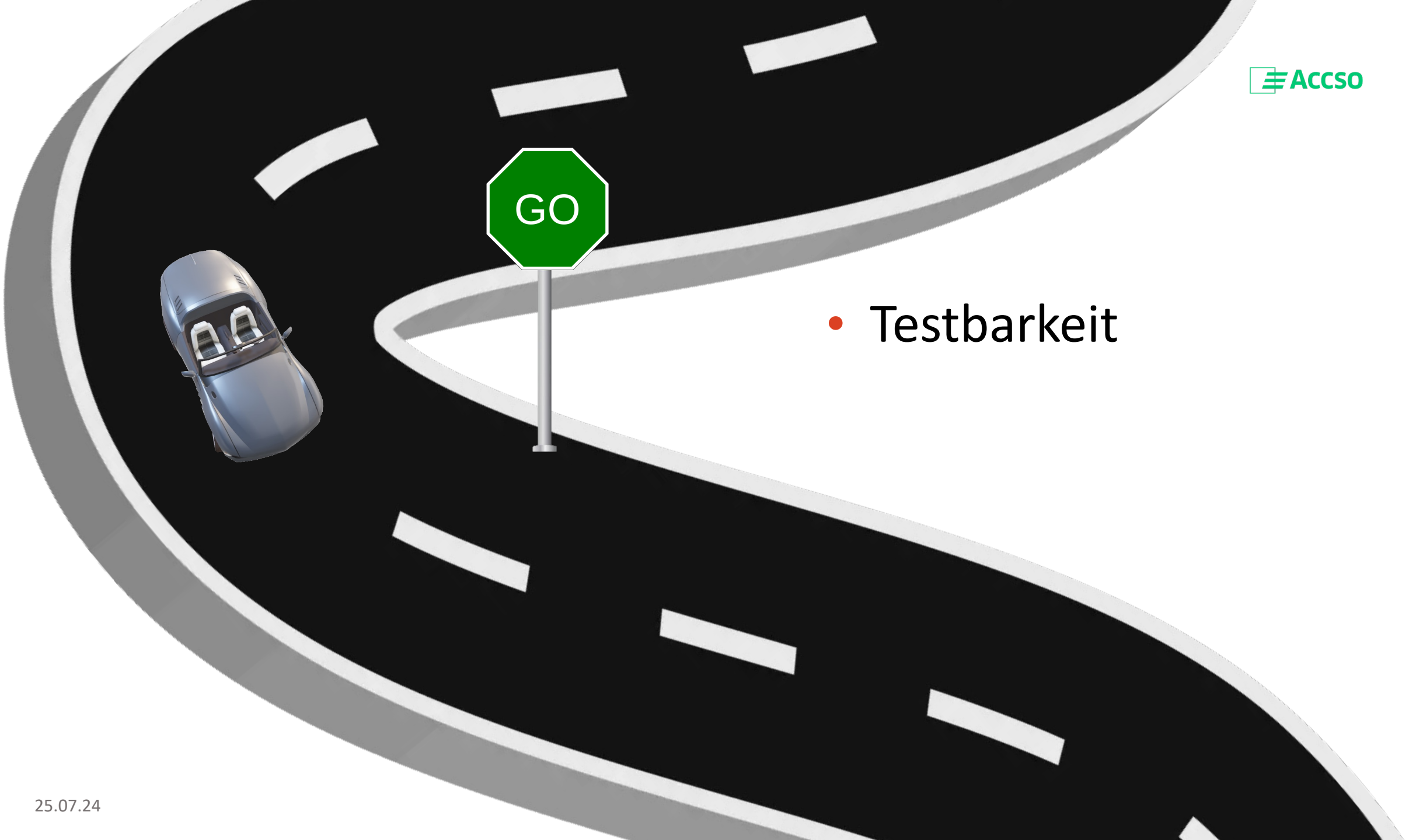
Accso

Der Stil der Präsentation

Vor- und Nachteile von den domänenzentrischen Architekturen

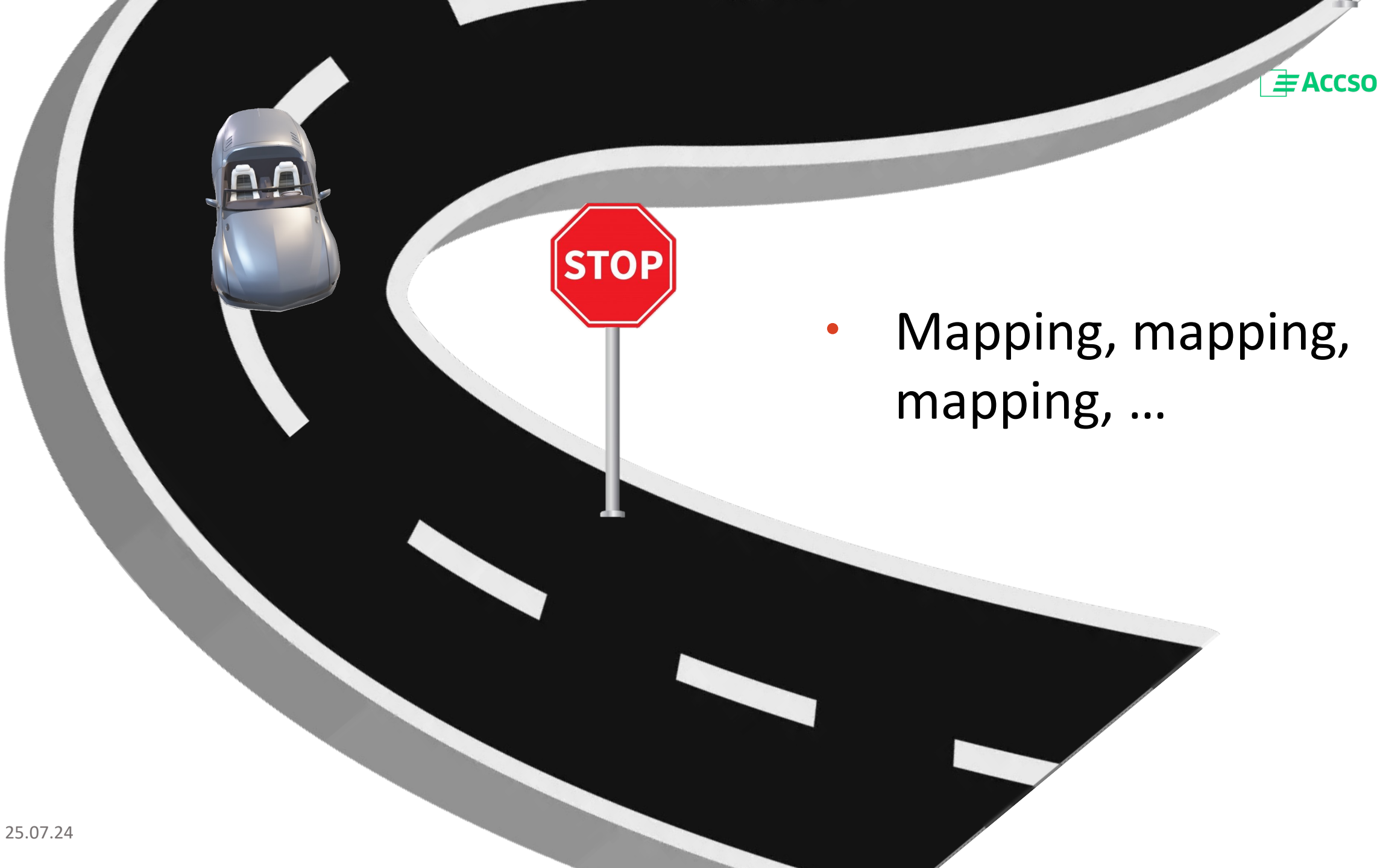


- Die genannten Architekturen stellen klare Richtlinien und Strukturen für die Platzierung bestimmter Komponenten bereit.

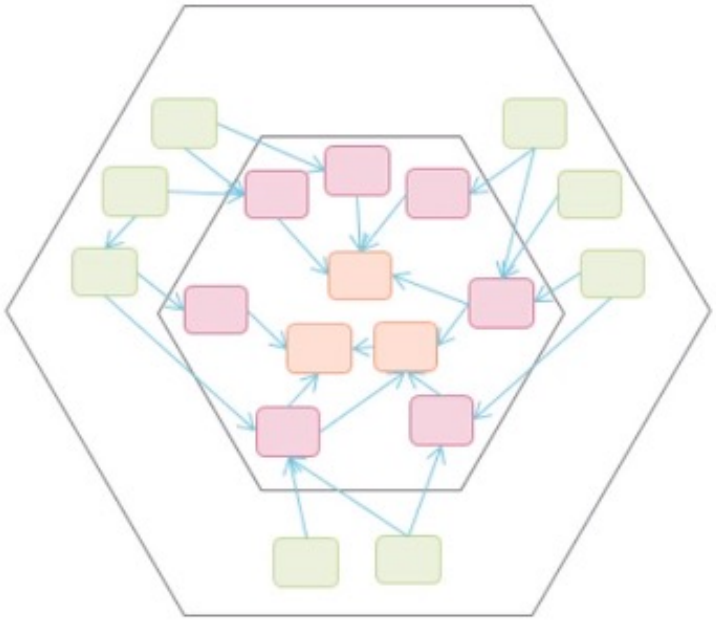


- Testbarkeit

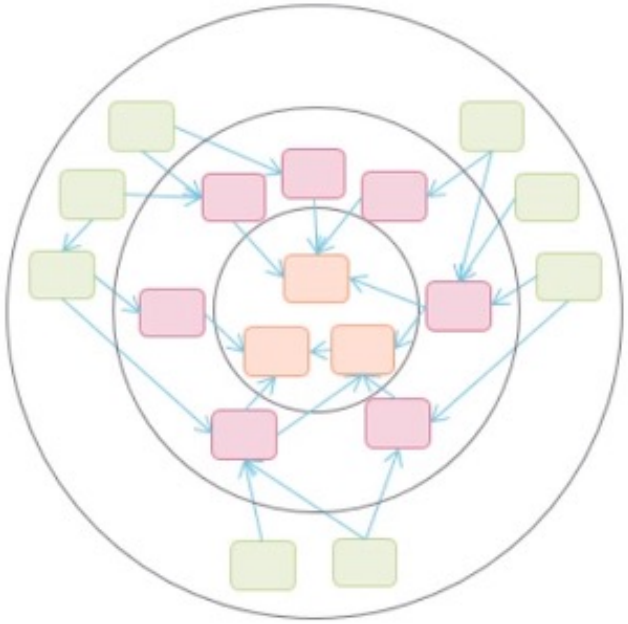
- Die Architekturen sind nicht so klar strukturiert, wie man es sich erhoffen würde. Sie bieten sowohl Struktur als auch Freiraum.



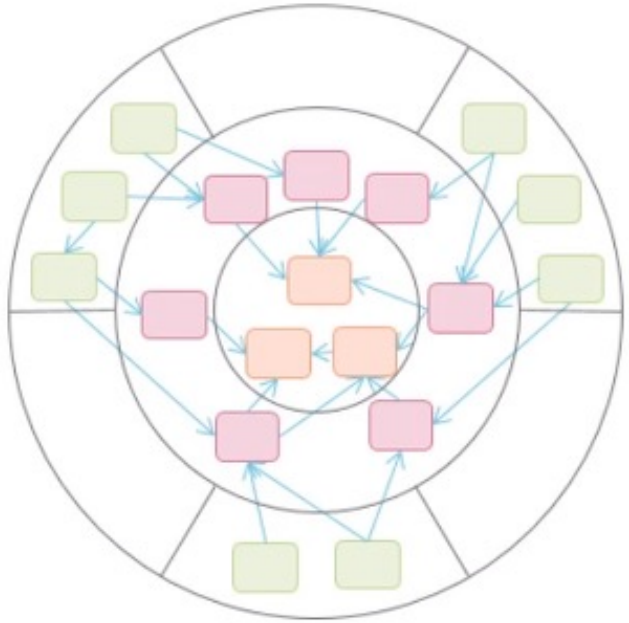
- Mapping, mapping, mapping, …



Hexagonal



Onion



Clean

Verwendet die Architektur, mit der ihr euch wohlfühlt!

Die wichtige Frage ist, wann verwende ich überhaupt eine domänenzentrische Architektur!

Wann domänenzentrische Architekturen einsetzen?!

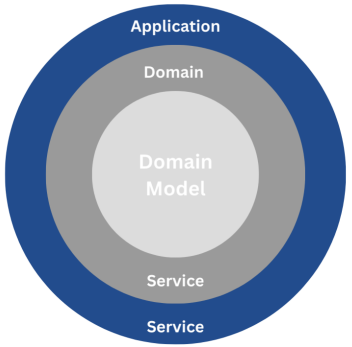
ACCSO

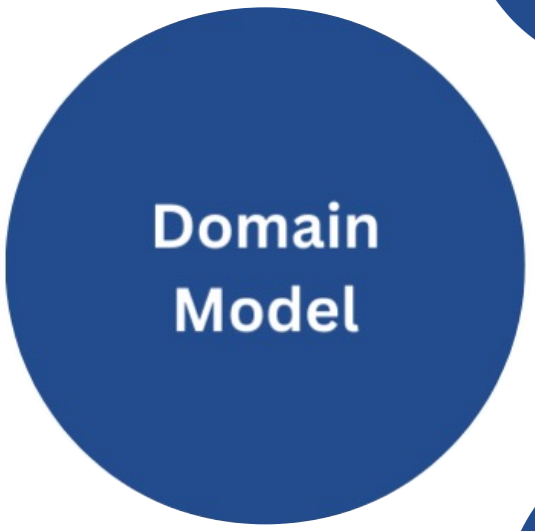
Die Implementierung von Fachlichkeit steht im Vordergrund

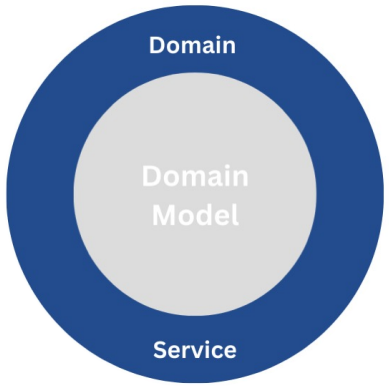
Es handelt sich nicht um: eine triviale CRUD-Anwendung, hardwarenahe Software, Bibliotheken usw.

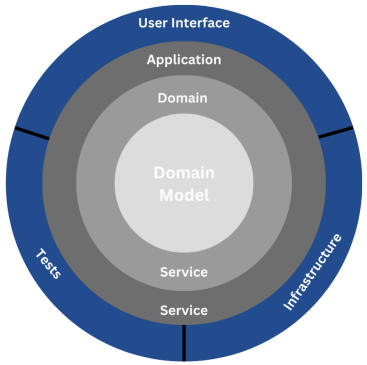
Die Software soll langlebig sein und darin wird auch investiert

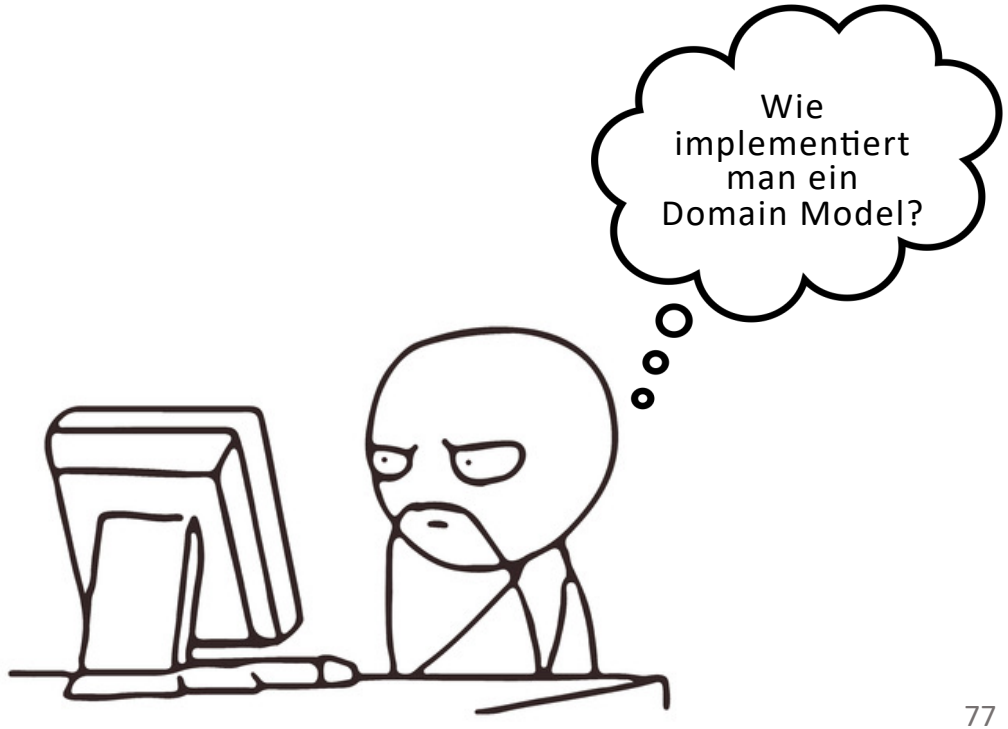
Es handelt sich nicht um: Prototypen oder *Einweg-Messeprodukte*


Application
Domain
Domain Model
Service
Service


Domain Model

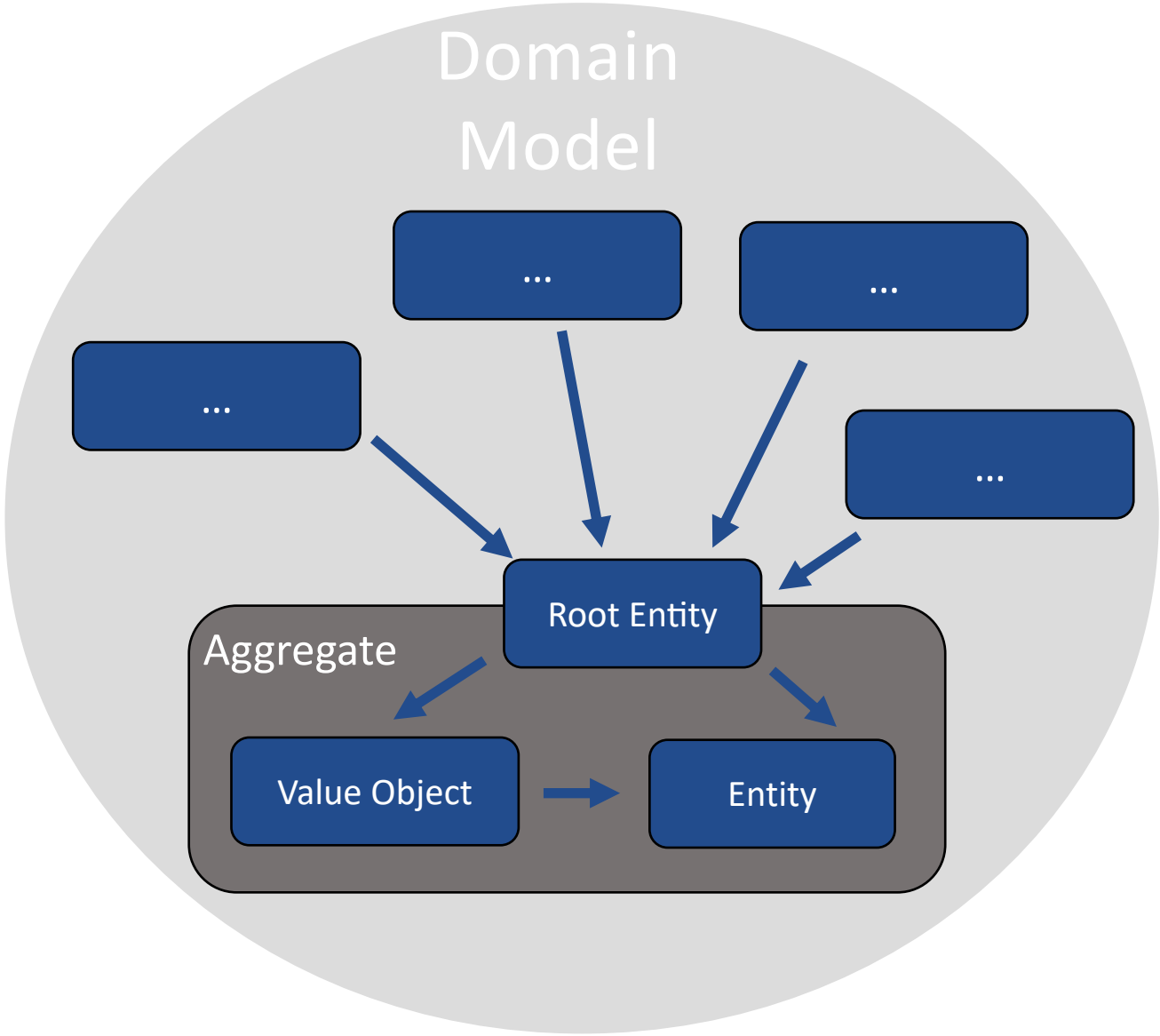

Domain
Domain Model
Service


User Interface
Application
Domain
Domain Model
Service
Service
Tests
Infrastructure


Wie implementiert man ein Domain Model?

Domain Model Implementierung – DDD Taktisches Design

- Muster aus dem Taktischen Design
  - Value Object
  - Entity
  - Aggregate
  - …



Value Object

ACCSO

```
public record Anzahl(int anzahl) {
    public Anzahl {
        if(anzahl < 0) {
            throw new IllegalArgumentException("Die Anzahl darf nicht negativ sein.");
        }
    }

    public Anzahl erhoeheUm(Anzahl anzahl) {
        return new Anzahl(this.anzahl + anzahl.anzahl);
    }

    //...
}
```

Eigenschaften:

- Unveränderlich
- Vergleich über die Attribute
- Reichhaltig an Fachlichkeit
- Selbstvalidierend

Entity

ACCSO

```
public class Warenkorbzeile {
   private final WarenkorbzeileID warenkorbzeileID;
   private final ArtikelID artikelID;
   private Anzahl anzahl;
   private final Preis preis;
   private final Anzahl maxArtikelAnzahl;

   public Warenkorbzeile(WarenkorbzeileID id, ArtikelID artikelId, Anzahl anzahl, Preis preis, Anzahl maxArtikelAnzahl) {
      //..
      validiere();
   }

   public void erhoeheUm(Anzahl anzahl) {
      this.anzahl = this.anzahl.erhoeheUm(anzahl);
      validiere();
   }

   private void validiere(){
      // Prüfe Pflichtfelder auf not null
      //..
      if(getAnzahl().istGroesserAls(maxArtikelAnzahl)) {
         throw new IllegalStateException("Maximale Anzahl von Artikel ist überschritten.");
      }
   }
}
```

Eigenschaften:

- Hat einen eigenen Lebenszyklus
- Vergleich über Id
- Reichhaltig an Fachlichkeit
- Selbstvalidierend (ggf. abhängig vom Status)
- …

Aggregate

```
public class Warenkorb {
    private final WarenkorbID warenkorbID;
    private final KundeID kundeID;
    private final List<Warenkorbzeile> warenkorbzeilen;
    private Preis gesamtPreis;
    private final Preis maxEinkaufswert;

    public Warenkorb(WarenkorbID warenkorbId, KundeID kundeID, Preis maxEinkaufswert) {
        //..
        this.validiere();
    }

    public void entferne(Artikel artikel) {
        Warenkorbzeile zeileMitArtikel = findeZeileZu(artikel.artikelID());
        if (zeileMitArtikel != null) {
            Anzahl anzahlVonArtikel = zeileMitArtikel.getAnzahl();
            Preis gesamtpreisVonArtikel = berechneGesamtpreis(artikel.preis(), anzahlVonArtikel);
            warenkorbzeilen.remove(zeileMitArtikel);
            this.gesamtPreis = gesamtPreis.reduziereUm(gesamtpreisVonArtikel);
        }
        validiere();
    }

    private void validiere(){
        // Prüfe Pflichtfelder auf not null
        //..

        if (getGesamtPreis().istGroesserAls(maxEinkaufswert)) {
            throw new IllegalStateException("Maximaler Einkaufswert ist überschritten.");
        }
    }
}
```

Accso

Eigenschaften:

- Fasst mehrere Entities zusammen
- Ist eine fachlich motivierte Transaktionsgrenze
- Modellierung anhand Invarianten und Use Cases
- Ist eine Root – Entity als Einstiegspunkt
- …

Fazit

Accso

Die Trennung von fachlichem und technischem Code wirkt sich grundsätzlich positiv auf die Wartbarkeit der Software aus.

Das viele Mapping bedingt einen erhöhten Arbeitsaufwand.

Die Architekturmuster können nur selten 1:1 im Projekt übernommen werden. Anpassungen sind fast immer notwendig.

Keines dieser Architekturmuster steht im Wettbewerb zueinander.

Vielen Dank für Eure Aufmerksamkeit!

gulmariyam.yerzhanova@accso.de

https://accso.de/wissen/publikationen

https://github.com/gulmariyam/bachelorarbeit-yerzhanova

Accso – Accelerated Solutions GmbH

T | +49 6151 13029-0
E | info@accso.de
@ | www.accso.de

Hilpertstraße 12 | 64295 Darmstadt
Rahmhofstraße 2 | 60313 Frankfurt a. M.
Im Mediapark 6° | 50670 Köln
Balanstraße 55 | 81541 München
Clocktower, 302 | CV&A Waterfront, Cape Town 8002, ZA