Softwaretechnik

Prof. Dr. Raphael Herding

Wofür brauche ich Architektur?

Hierfür braucht man keine Architektur.

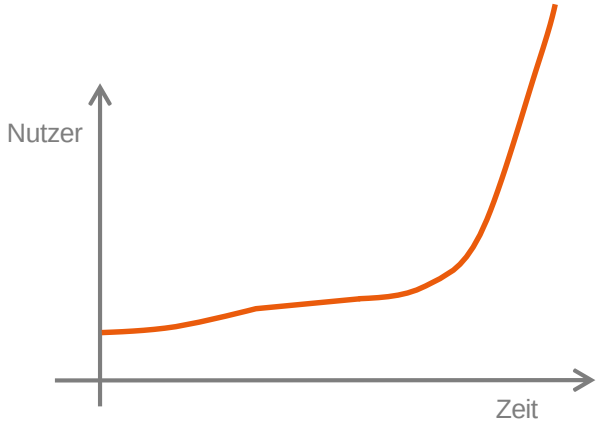
Hierfür schon.

… und hierfür braucht man Enterprise-Architektur

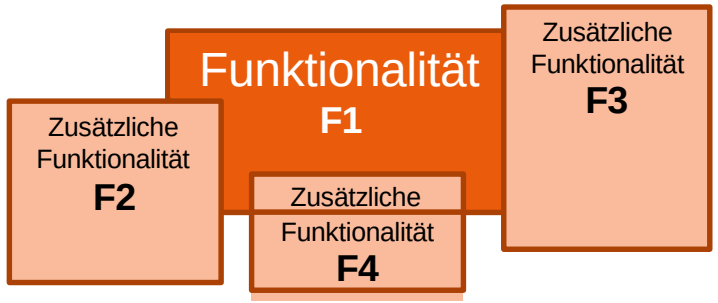
… sonst entsteht dies.

IT-Architektur -
vom Kleinen zum Großen

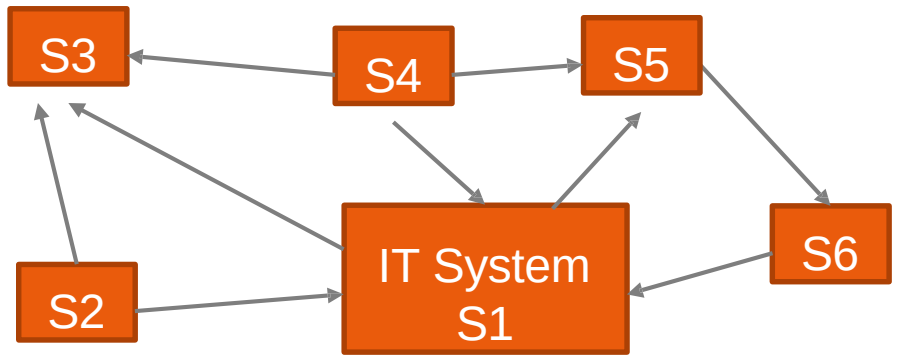
Anforderungen an IT-Systeme ändern sich über die Zeit!



(1) Anzahl Nutzer steigt drastisch

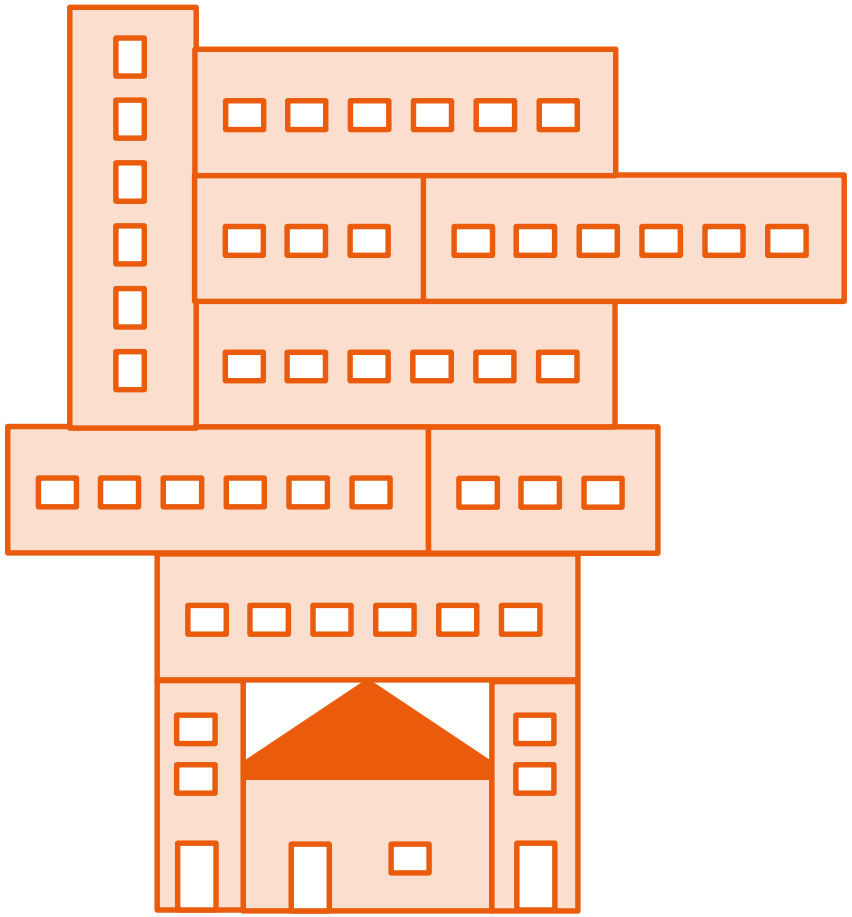


(2) Zahlreiche (unvorhergesehene) neue Funktionalitäten

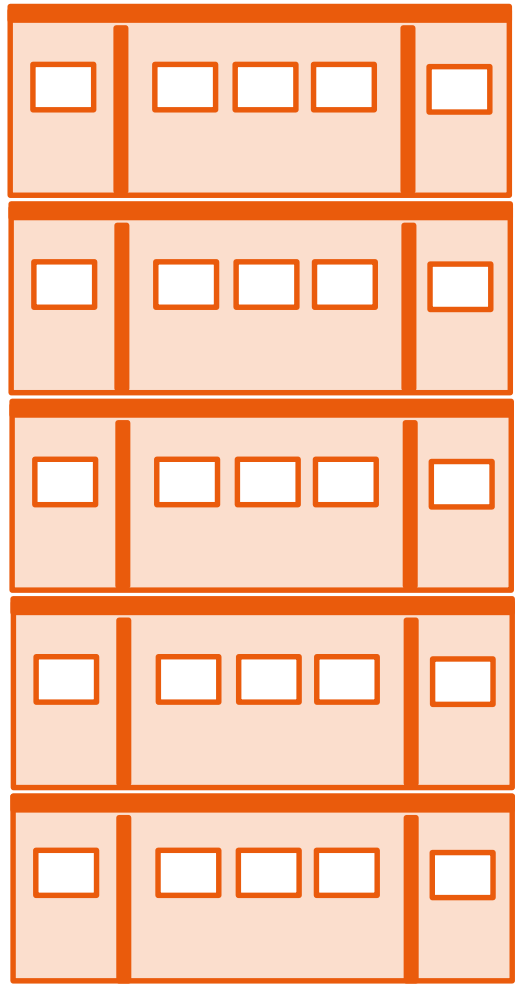


(3) Als "Standalone"-System geplante Anwendung wird über die Zeit in eine vernetzte IT-Landschaft eingebunden

Welches Design wähle ich zu Beginn?

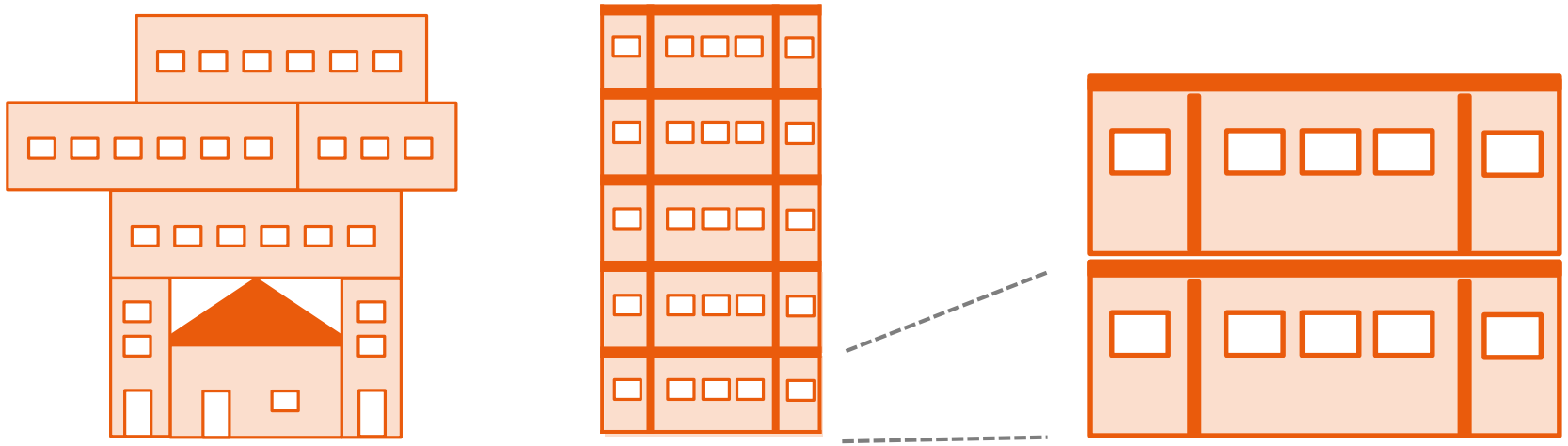


So?



… oder so?

EAM: Der Ausweg aus dem Dilemma



Das Problem ist meist **nicht die Technologie** …

- falscher Ziegeltyp
- Oracle statt Postgres
- .NET statt Java
- …

sondern die **Strukturen.**

Architekturmanagement hilft, die Strukturen der IT-Landschaft zu …

- planen
- überwachen
- transformieren

Agilität und Softwarearchitektur

Warum Architektur wenn sich doch alles so schnell ändert?

Agilität und Softwarearchitektur

- Warum Architektur wenn sich doch alles so schnell ändert?

- Es ist allgemein anerkannt, dass eine frühe Phase der agilen Prozesse darin besteht, eine umfassende Systemarchitektur zu entwerfen.

- Die Überarbeitung der Systemarchitektur ist in der Regel teuer, weil sie so viele Komponenten des Systems betrifft.

Softwarearchitektur und Granularität

- Architektur im **Kleinen**
  - Zerlegung eines einzelnes Programms in Komponenten (Komponentenarchitektur, Schichtenmodelle).

- Architektur im **Großen**
  - Architektur komplexer Unternehmenssysteme, die andere Systeme, Programme und Programmkomponenten umfassen.
  - Verteilung über verschiedene Computer und Dienste, die sich im Besitz verschiedener Unternehmen befinden und von diesen verwaltet werden.

- Vorteile einer Softwarearchitektur
  - Verbesserte Strukturierung und Organisation
  - Einfachere Stakeholder Kommunikation
  - Ermöglichen von weitreichender Systemanalyse
  - Förderung von Wiederverwendung

Architekturrepräsentationen

- Je nach Verwendungszweck können einfache **Blockdiagramme** ausreichen.
- Blockdiagramme haben jedoch **semantische Ausdrucksschwächen**
  - Zeigen weder die Art der Komponentenbeziehungen noch
  - die von außen sichtbaren Eigenschaften der Teilsysteme.
- Sie sind jedoch nützlich für die Kommunikation mit den Stakeholdern und für die

  Projektplanung.

- Für **mehr Ausdrucksstärke** können **UML Diagramme/Archimate, …** genutzt werden.
- Neben UML existieren andere Modellierungstechniken (jedoch außerhalb dieses Kurses).

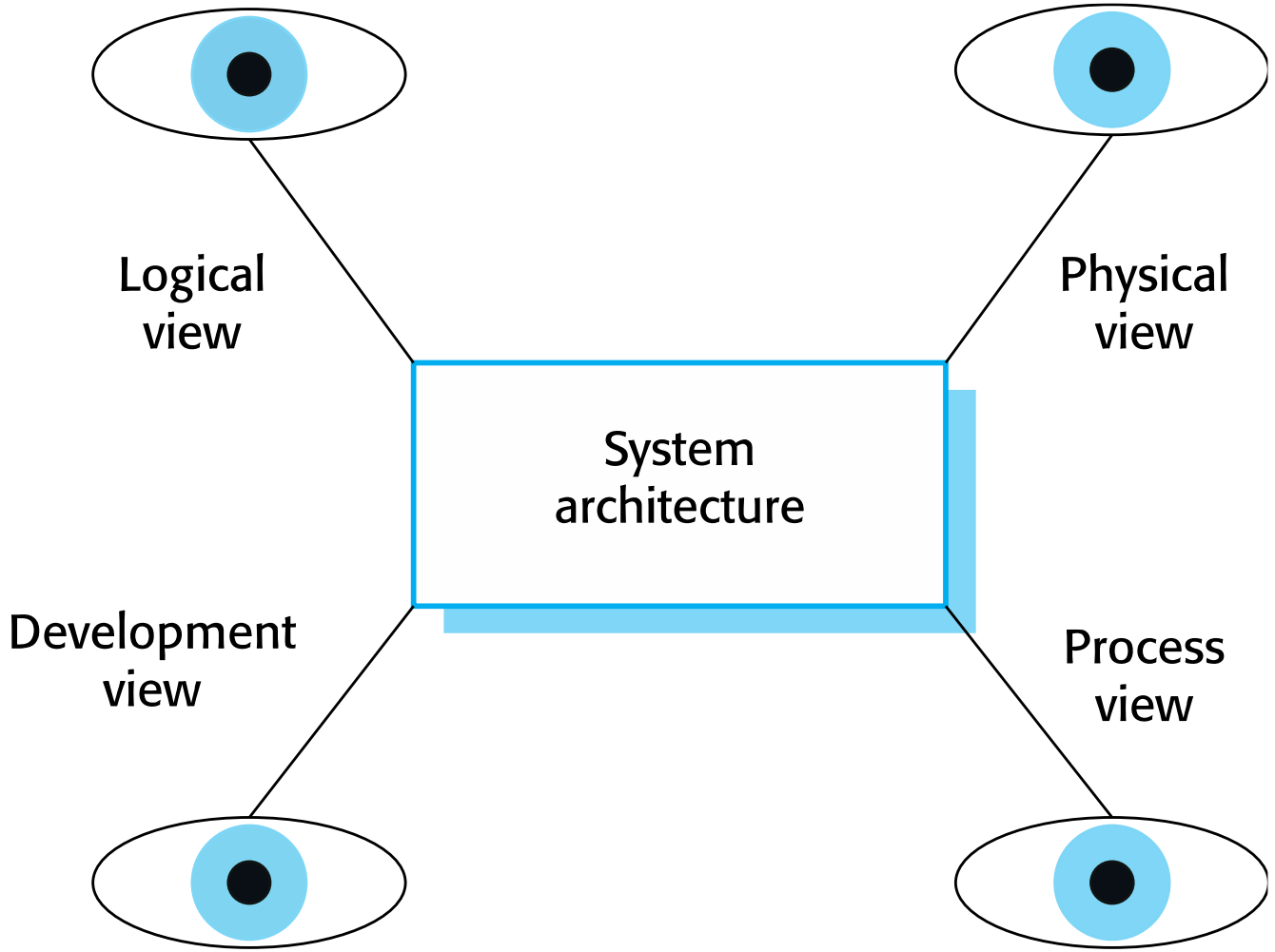
Architekturansichten

- Um Architekturmodelle besser verstehen zu können, werden unterschiedliche Ansichten benötigt.
- Typische Fragen bei der Erstellung der Ansichten:
  - Welche Sichten oder Perspektiven sind beim Architekturentwurf sinnvoll?
  - Welche Notationen sollten verwendet werden?

- Jedes Architekturmodell kann mehrere **Ansichten** oder **Perspektiven** besitzen
  - Modulzerlegung eines Systems
  - Interaktion der Laufzeitprozesse
  - verschiedenen Systemkomponenten, die über ein Netzwerk verteilt sind
  - Komponenten- und Servicedesign
  - …
- Es müssen in der Regel mehrere Sichten mit unterschiedlichen Abstraktionsebenen erstellt werden,

  um so die Gesamtarchitektur sinnvoll gestalten zu können.

Architekturansichten



Architekturansichten
Die 4+1 Ansicht

1. **Logische Ansicht**
   - stellt wichtige Abstraktionen im System als Objekte oder Objektklassen dar.
   - z.B. durch **UML Komponentendiagramm**
2. **Prozessansicht**
   - zeigt, wie das System zur Laufzeit aus interagierenden Prozessen zusammengesetzt ist.
   - z.B. durch **UML-Sequenzdiagramme oder UML-Ablaufdiagramme**
3. **Entwicklungsansicht**
   - zeigt, wie die Software entwickelt wird.
   - Z.b. **Schichtenarchitektur**, **UML-Komponentendiagramm**, Service-Modell,…
4. **Physische Ansicht**
   - zeigt, wie die Softwarekomponenten auf die physikalische Hardware angeordnet wird
   - Z.B. **UML-Verteilungsdiagramm**, Deployment-Diagramme, Staging-Diagramme, …

- (**+1**) Stets **bezogen auf Anwendungsfälle oder Szenarien** des
  - Z.b. **UML-Anwendungsfalldiagramm, UML-Interaktionsdiagramme, …**

Lab

- Entwickeln Sie in Gruppen eine **physische Sicht** auf das bisher entwickelte Mentcare-System.
- Nutzen Sie dazu auch das Kontextmodell als Hilfe.

Designprinzipien

- **Richtlinien** zur Erstellung guter, wartbarer und robuster Softwaresysteme, die die Komplexität reduzieren und die Zusammenarbeit erleichtern.
- Ziele
  - **Wartbarkeit**: Ein gutes Design erleichtert die Wartung und Anpassung der Software über die Zeit.
  - **Lesbarkeit**: Der Code wird für andere Entwickler und das zukünftige Ich leichter verständlich.
  - **Flexibilität und Skalierbarkeit**: Systeme können leichter erweitert und an neue Anforderungen angepasst werden.
  - **Reduzierung von Komplexität**: Komplexe Systeme lassen sich in überschaubare, unabhängige Teile zerlegen.
  - **Wiederverwendbarkeit**: Komponenten können in verschiedenen Kontexten wiederverwendet werden.

Designprinzipien

- Wichtige Designprinzipien
  - **Separation of Concerns**
  - **SOLID**
  - **KISS** Principle (Keep it Simple, Stupid)
  - **YAGNI** (You Ain't Gonna Need It)
  - **Clean Architecture**
  - **Modularität** und **Abstraktion**
  - **Dependency Injection (DI)**
  - **Inversion of Control (IoC)**

Designprinzipien
Separation of Concerns

- Separation of Concerns ist eines der grundlegendsten Prinzipien in der

  Softwareentwicklung.

- Programm besteht nicht aus einem einzelnen Block
  - Mischung von Ausgabe und Eingabe
  - Mischung von Datenabfrage und Businesslogik
  - Mischung von funktionalem und technischem Code
  - …
- Zerlegen Sie den Code in kleine Teile, die jeweils eine bestimmte Aufgabe (Concern)

  erfüllen.

Lab

Lab c-01 Refactoring

Designprinzipien
SOLID

- Sammlung von **5 Prinzipien** für strukturiertes, wartbares objektorientiertes Design.

***S**RP*: Single-Responsibility-Prinzip
- Eine Klasse oder ein Modul sollte nur **einen einzigen Grund für eine Änderung** haben, also genau **eine klar abgegrenzte Aufgabe** erfüllen.

***O**CP*: Open-Closed-Prinzip
- Softwareeinheiten sollen **offen für Erweiterungen**, aber **geschlossen für Änderungen** sein, sodass neue Funktionen ohne Eingriff in bestehenden Code ergänzt werden können.

***L**SP*: Liskov'sche Substitutions-Prinzip
- **Unterklassen müssen sich wie ihre Basisklasse verhalten**, sodass Objekte der Basisklasse problemlos durch Objekte der Subklasse ersetzt werden können.

***I**SP*: Interface-Segregation-Prinzip
- **Viele kleine, spezifische Schnittstellen** sind besser als ein großes Interface, damit Clients nur die Methoden implementieren müssen, die sie tatsächlich benötigen.

***D**IP*: Dependency-Inversion-Prinzip
- **Höherwertige Module** sollen nicht von **niedrigwertigen Modulen**, sondern beide von **Abstraktionen** abhängen, um flexible und entkoppelte Architekturen zu ermöglichen.

Lab

Lab c-02

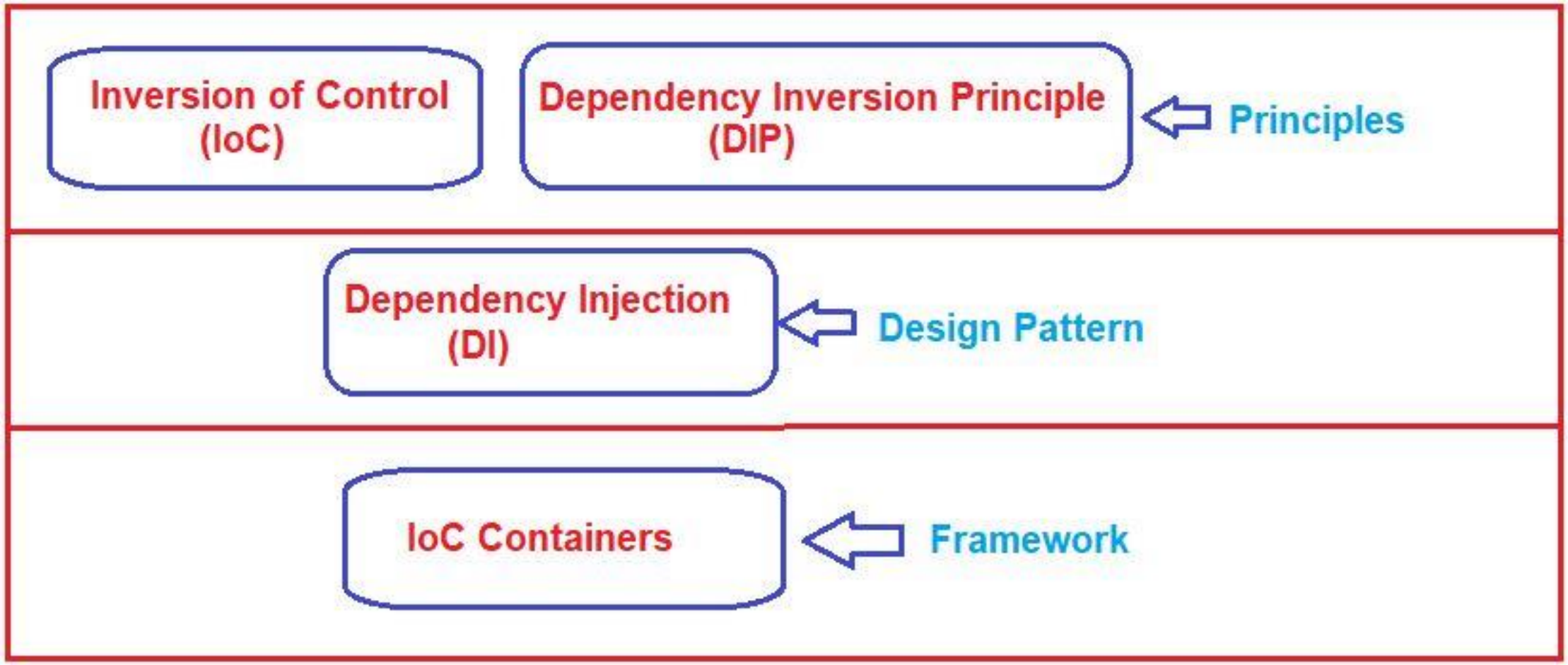
Designprinzipien

- Wichtige Designprinzipien
  - Separation of Concerns
  - SOLID
  - **KISS** Principle (Keep it Simple, Stupid)
    - fordert, dass Systeme und Lösungen so einfach wie möglich gestaltet werden, um sie leicht verständlich und wartbar zu halten.
  - **YAGNI** (You Ain't Gonna Need It)
    - besagt, dass Funktionalität nur implementiert werden sollte, wenn sie tatsächlich gebraucht wird, um unnötige Komplexität zu vermeiden
  - **Clean Architecture**
    - Design-Ansatz, der die Geschäftslogik von externen Abhängigkeiten isoliert und so die Wartbarkeit und Anpassbarkeit fördert.
  - **Modularität** und **Abstraktion**
    - Systeme sollten in unabhängige Module aufgeteilt und durch Abstraktionen zusammengehalten werden.

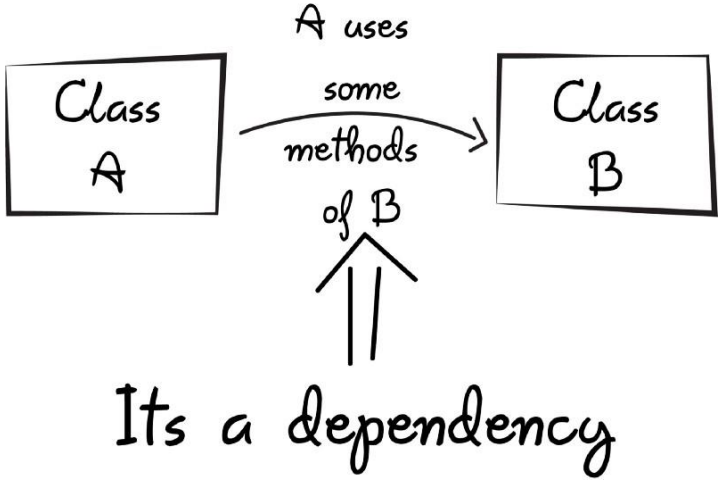
Lab

Lab c-03

Designprinzipien
Dependency Injection und Inversion of Control



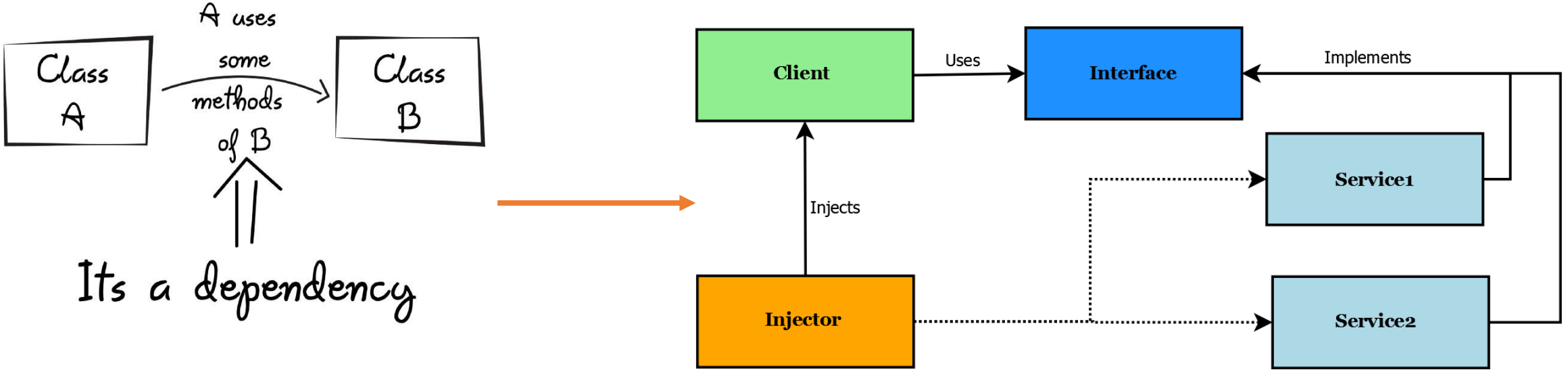
Designprinzipien
Dependency Injection und Inversion of Control



- Verwaltet Abhängigkeiten im Programmcode

Designprinzipien

Dependency Injection und Inversion of Control



- Verwaltet Abhängigkeiten im Programmcode
- Injector (Dependencymanagement Framework) instanziiert Objekte.
- Client hängt **nur vom Interface**, **nicht** von der **Klasse** ab.
  - Nötig, um später testbaren Code zu schreiben!

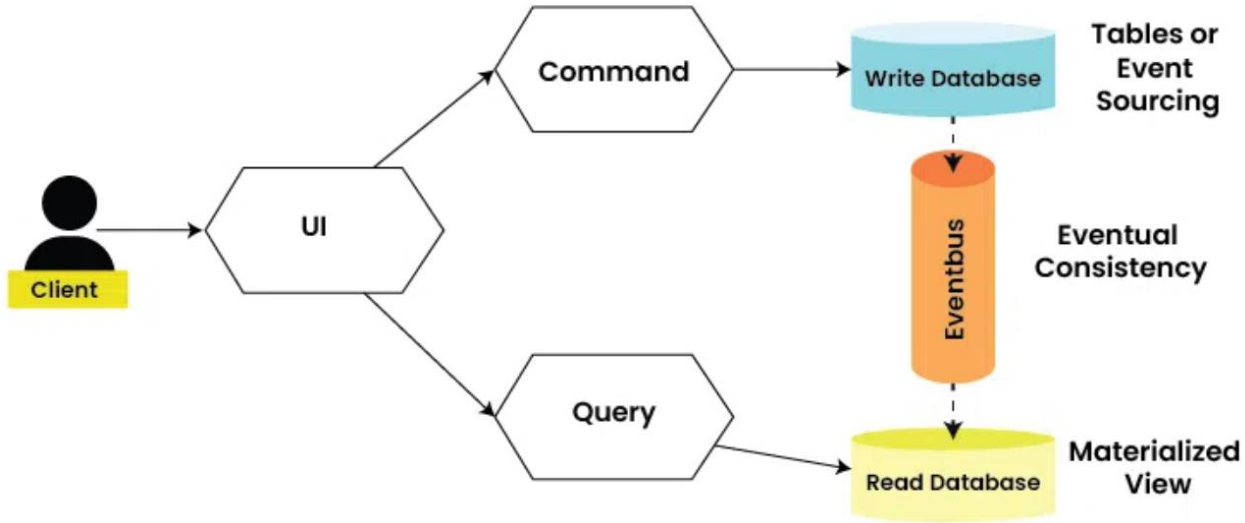
Lab

Lab c-04

Architekturmuster

- **Architekturmuster** sind Musterarchitekturen, die in bestimmten Fällen wiederverwendet werden können (vergleichbar mit Fertighaus aus Katalog).
- Sie wurden in verschiedenen Umgebungen (in der Praxis) erprobt und getestet.
- Wichtige Architekturmuster
  - Command Query Responsibility Segregation (CQRS)
  - Repository Pattern
  - Layered Architecture Pattern
  - Client/Server Pattern
  - Peer-to-Peer Pattern
  - Master/Slave Pattern
  - Pipe-Filter Pattern
  - Event Driven Architecture (Broker Pattern)
  - Event Driven Architecture (Event-Bus Pattern)
  - Event Driven Architecture (Massage Queuing Pattern)

Architekturmuster
Command Query Responsibility Segregation (CQRS)

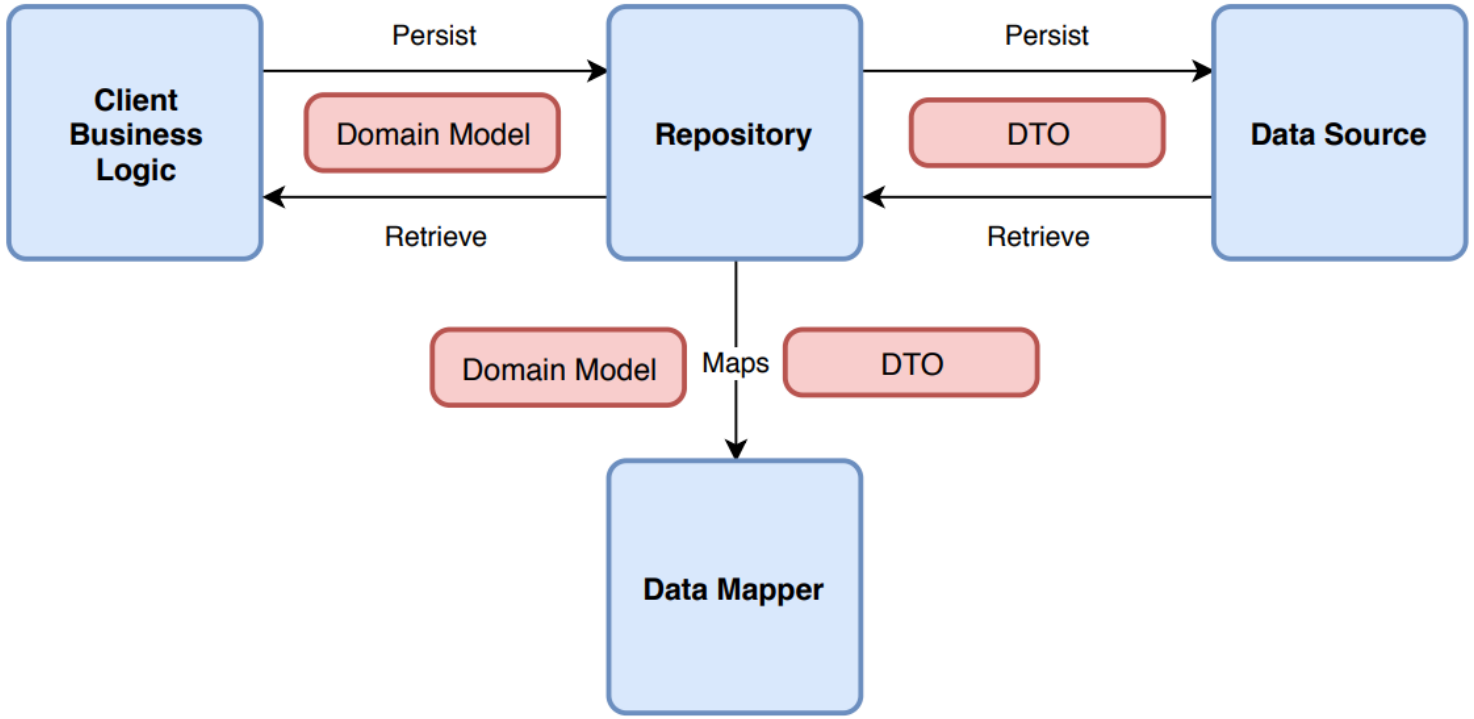


- In großen Systemen wird das **gemeinsame Lesen** und **Schreiben** von Daten zum Engpass.
- CQRS löst das und trennt Leseoperationen von Schreiboperationen.
- **Commands**: Implizieren einen Zustandsübergang und sind demzufolge Schreiboperationen.
- **Queries**: Beschreibt die Informationsbeschaffung und somit Leseoperationen.

Lab

Lab c-05

Architekturmuster
Repository Pattern



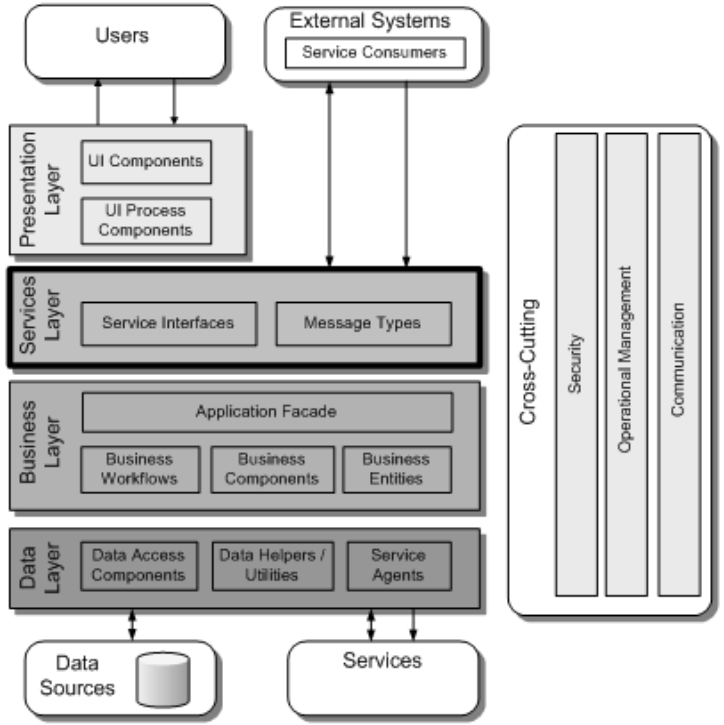
- Trennt die Businesslogik von der Datenhaltung.
- Kommunikation über Fachobjekte (Data Transfer Objects - DTOs).
- Repository bietet Funktionen zum Lesen, Schreiben und Bearbeiten der Daten an.

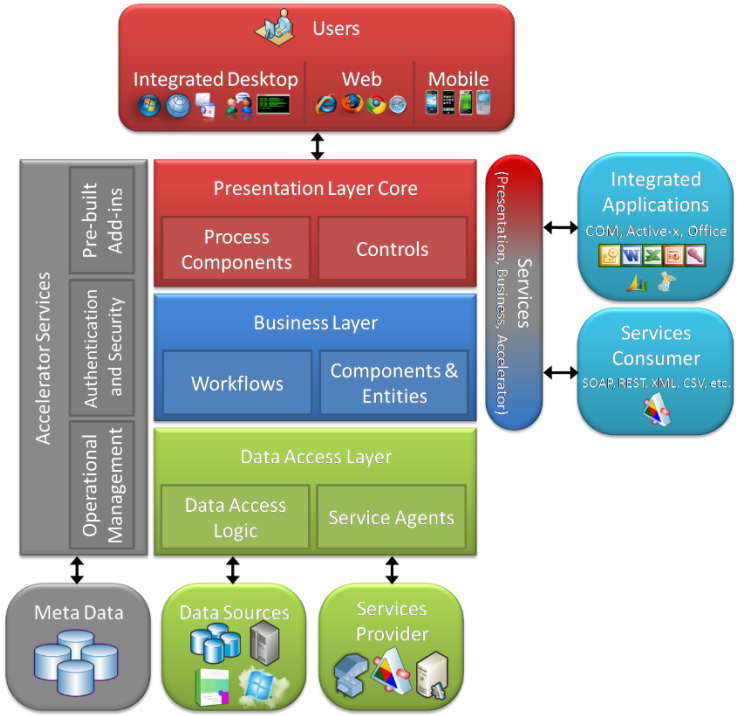
Lab

Lab c-06

Architekturmuster

Layered Architecture Pattern





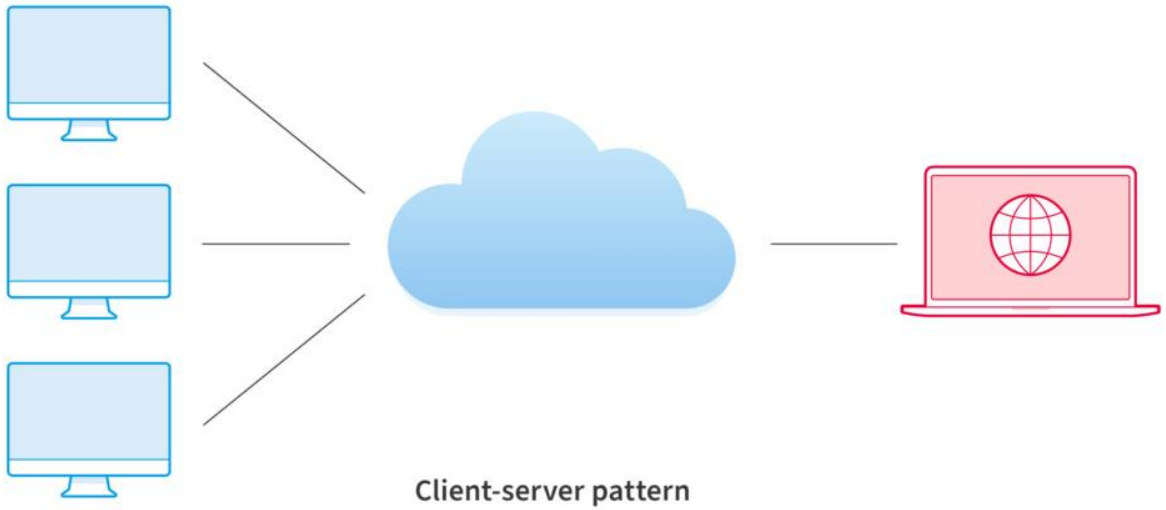
- Organisiert das System in eine Reihe von Schichten (sowie Angabe verfügbarer Dienste)
- Unterstützt die schrittweise Entwicklung von Teilsystemen in verschiedenen Schichten
- Bei Änderung der Schnittstelle einer Schicht => nur Änderung der benachbarten Schicht nötig.
- Weitere Details hier: https://www.oreilly.com/library/view/software-architecture-patterns/9781491971437/ch01.html

https://nix-united.com/blog/10-common-software-architectural-patterns-part-1/

Lab

- Nehmen Sie eine Komponente aus ihrem Kontextmodell und erstellen sie dazu ein Layered Architecture Diagramm.

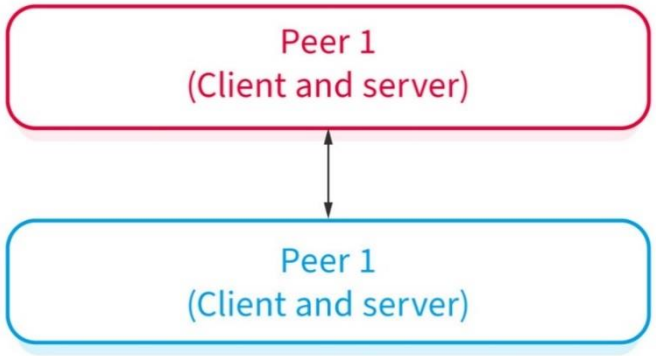
Architekturmuster
Client/Server Pattern



- Server bietet (provided) Dienstleistung den Clients (consumes) an.
- Clients senden Anfragen (Request) zur Nutzung der Dienstleistung.
- Server bearbeitet die Dienstanfrage und Antwortet (Response).
- Server ist passiv und muss stets vom Client zuerst aktiv angefragt werden.
- Response kann synchron oder asynchron durchgeführt werden.

https://nix-united.com/blog/10-common-software-architectural-patterns-part-1/

Architekturmuster
Peer-to-Peer Pattern



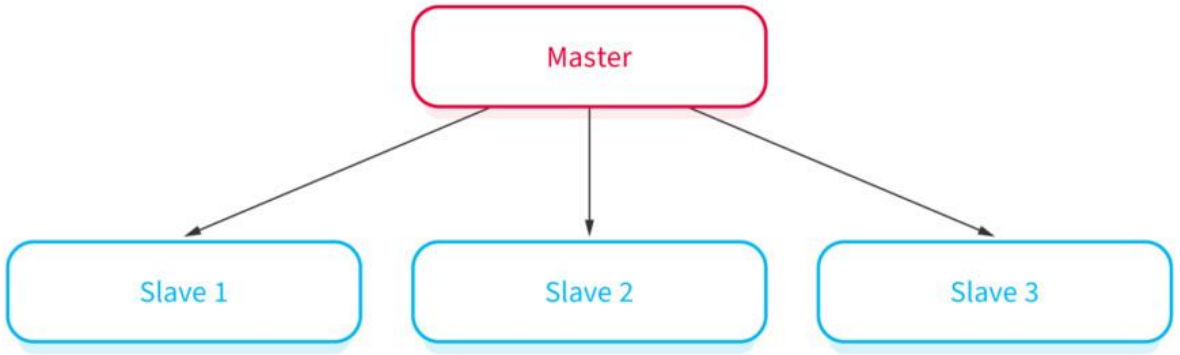
Peer-to-peer pattern

- Anders als beim Client/Server Pattern kann jeder Client aktiv Anfragen an andere senden.
- Jeder Teilnehmer hat die gleichen Berechtigungen (gleichgestellt).
- Keine zentrale Koordinationsinstanz vorhanden. Koordination erfolgt dezentral.
- Jeder Teilnehmer kann Client (consumer) oder Server (provider) sein.
- Kommunikationsaufwand steigt mit jedem neuen Teilnehmer.

https://nix-united.com/blog/10-common-software-architectural-patterns-part-1/

Architekturmuster

Master/Slave Pattern bzw. Leader/Follower

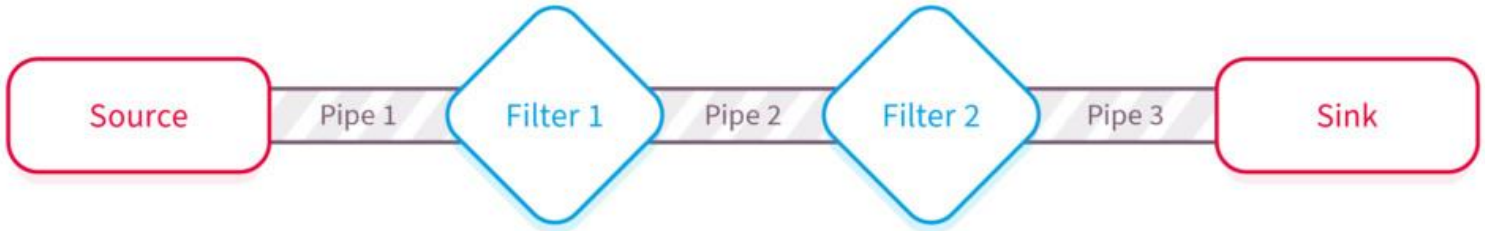


Master-slave pattern

- Master/Leader steuert (oder startet) Slaves/Follower.
- Slaves bearbeiten Aufträge und melden Ergebnis zurück an Master.
- Ausfälle von Slaves beeinträchtigen das System nicht. Master ist Engpass.
- Nur Master kann Operationen annehmen.
- Koordinationsverfahren (z.B. Daten Sync) können über Master/Slave Patterns umgesetzt werden.
- Verteitle und skalierbare Berechnungen können durch Master/Slave umgesetzt werden.

https://nix-united.com/blog/10-common-software-architectural-patterns-part-1/

Architekturmuster
Pipeline Pattern

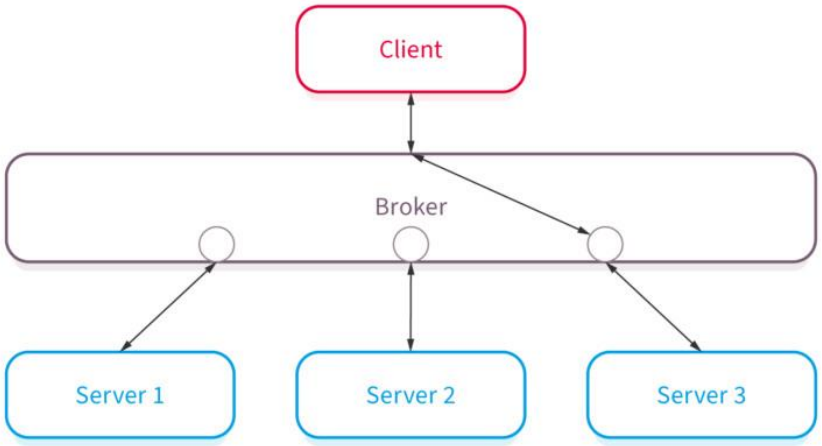


Pipe-filter pattern

- Zur Strukturierung von Systemen, die einen kontinuierlichen Datenstrom (Data Stream) erzeugen und verarbeiten.
- Jeder Verarbeitungsschritt wird von einer Filterkomponente (Filter) umschlossen.
- Filter enthält Filterbedingungen und ggf. Teile der Verarbeitungslogik.
- Daten werden durch Pipes zu den jeweiligen Filtern geleitet.
- Vorgelagerte Filter können (müssen aber nicht) Daten für nachgelagerte Filter anpassen.
- Pipes können für die Pufferung oder für Synchronisationszwecke verwendet werden (Request-Pipeline).

https://nix-united.com/blog/10-common-software-architectural-patterns-part-1/

Architekturmuster
Broker Pattern

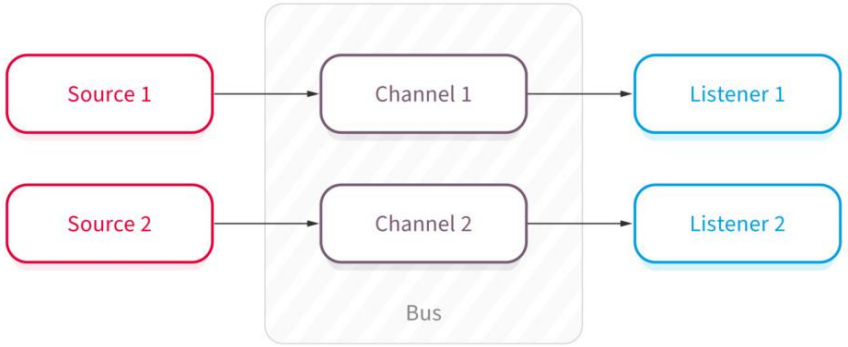


Broker pattern

- Broker koordiniert Kommunikation (Anfragen, Antworten) zwischen verschiedenen Clients und Servern.
- Server registriert angebotene Dienste beim Broker.
- Clients können angebotene Dienste beim Broker anfragen und werden durch diesen weitergeleitet.
- Sinnvoll, wenn die Clients die Server nicht „kennen“ sollen.
- Broker bildet den Engpass.

https://nix-united.com/blog/10-common-software-architectural-patterns-part-1/

Architekturmuster
Event-Bus Pattern

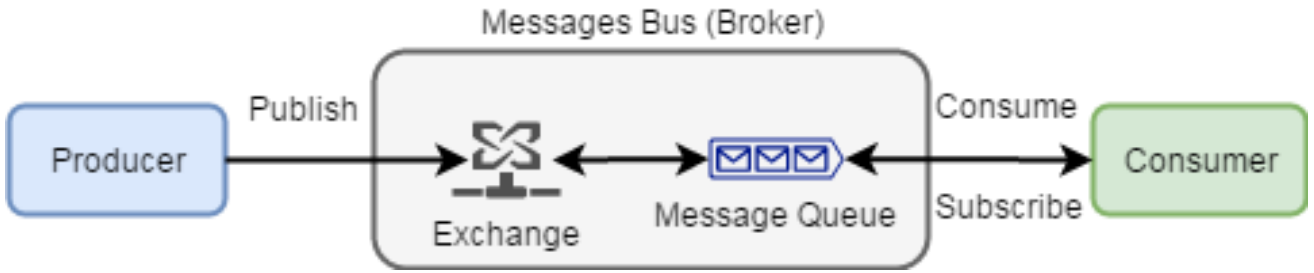


Event-bus pattern

- Ereignisorientierte Kommunikation. Zentraler Event-Bus vermittelt Ereignisse.
- Event-Bus enthält themenspezifische Channels.
- Quelle (Source bzw. Provider) kann Nachrichten in Channel einstellen.
- Konsumenten (Listener bzw. Subscriber) können Nachricht empfangen und verarbeiten.
- Sinnvoll wenn Provider Consumer nicht kennen soll (z.B. Notification senden).
- Sinnvoll für One-to-Many Kommunikation.
- Weitere Details hier https://www.oreilly.com/library/view/software-architecture-patterns/9781491971437/ch02.html#idm46407727806784

https://nix-united.com/blog/10-common-software-architectural-patterns-part-1/

Architekturmuster
Message Queuing Pattern



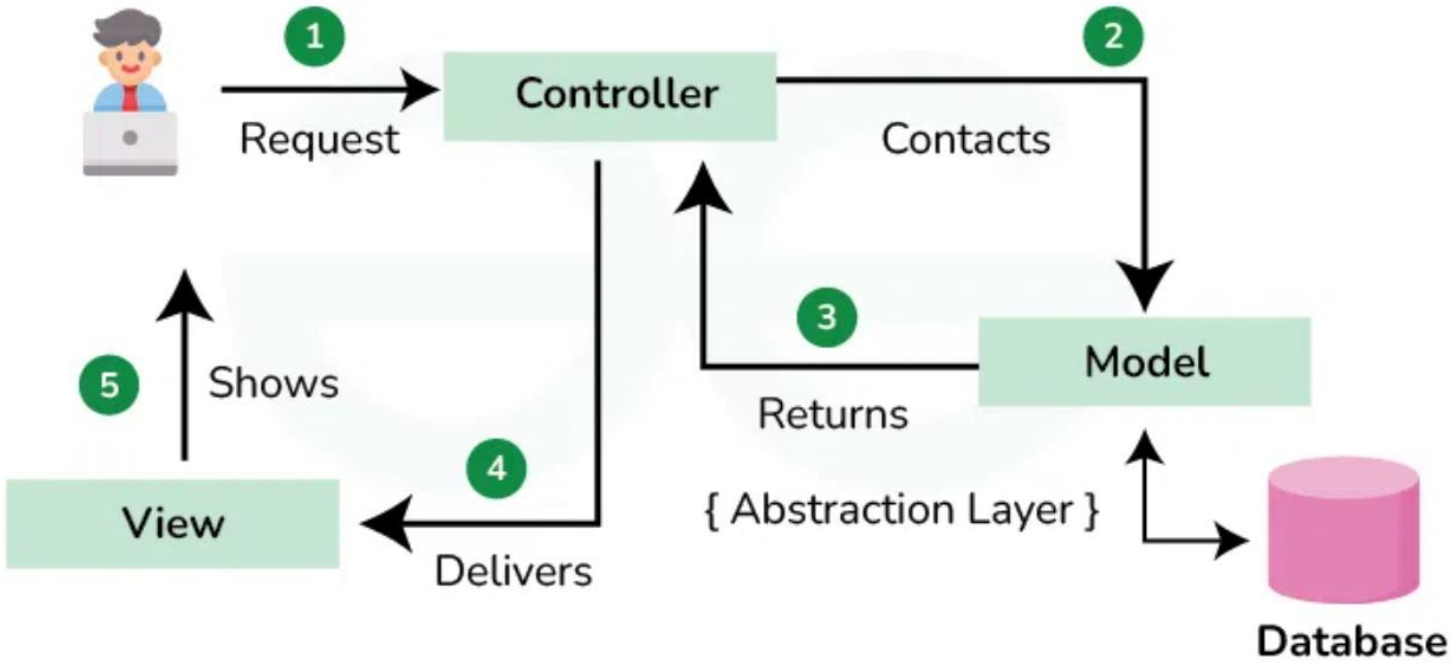
- Producer veröffentlicht (publish) Nachricht in der Queue.
- Consumer registriert (subscribe) sich auf eine Queue.
- Nachrichten werden vom Message Broker in der Queue verwaltet und zugestellt.
- Verschiedene Queuing Strategien möglich (FIFO, LIFO, Priority, …).
- Entkoppelt Producer von Consumer.
- Many-to-One Kommunikation (nur **ein** Consumer erhält die Nachricht aus der Queue)

https://nix-united.com/blog/10-common-software-architectural-patterns-part-1/

Standardarchitekturen

- Model View Controller Architektur
- Service-orientierte Architektur
- Hexagonale Architektur

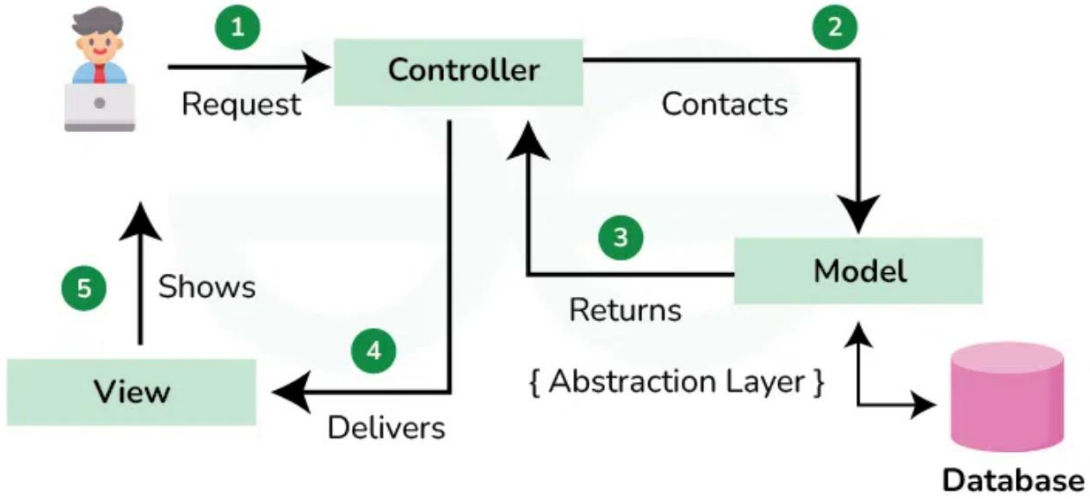
Standardarchitekturen
Model View Controller



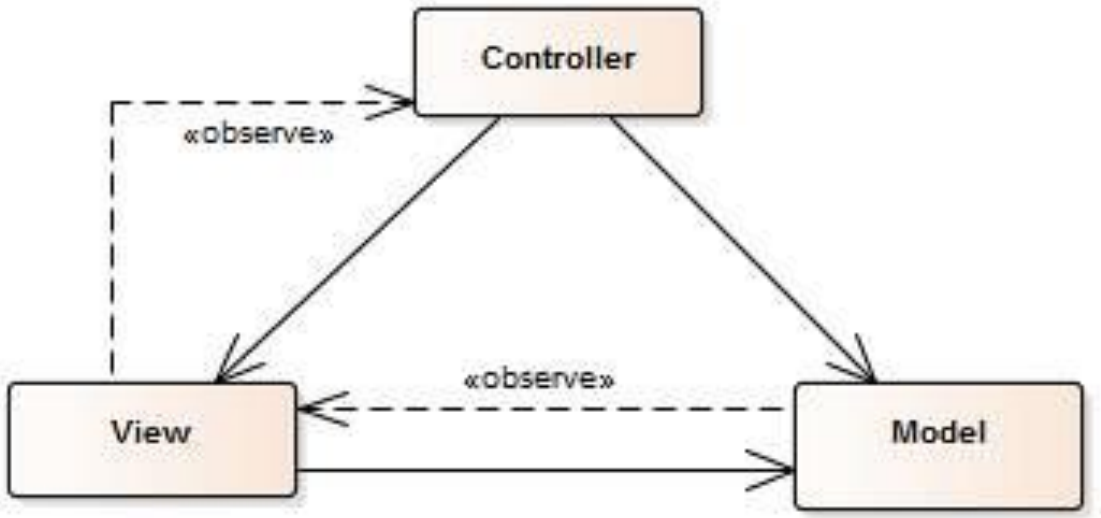
- Unterteilt die Programmlogik einer Oberfläche in die Komponenten Model, View und Controller.
- **Model**: verwaltet die Daten und Regeln der Anwendung.
- **View:** stellt die Daten dar
- **Controller**: interagiert mit dem Benutzer.

Standardarchitekturen
Model View Controller

- **Zweck**
  - Benutzeroberflächen müssen häufig geändert werden.
  - Unterschiedliche Benutzeroberflächen müssen unterstützt werden.
  - Datenmodell ist typischerweise stabil.
- **Lösung**
  - Die Anwendung wird in die Komponenten **Model** (Datenmodell), **View** (Ausgabekomponenten) und **Controller** (Eingabekomponenten) unterteilt
  - **mehrere Views** können das gleiche **Model** verwenden (z.B. mobile Ansicht, Desktop, Web UI, ...)



Standardarchitekturen
Model View Controller



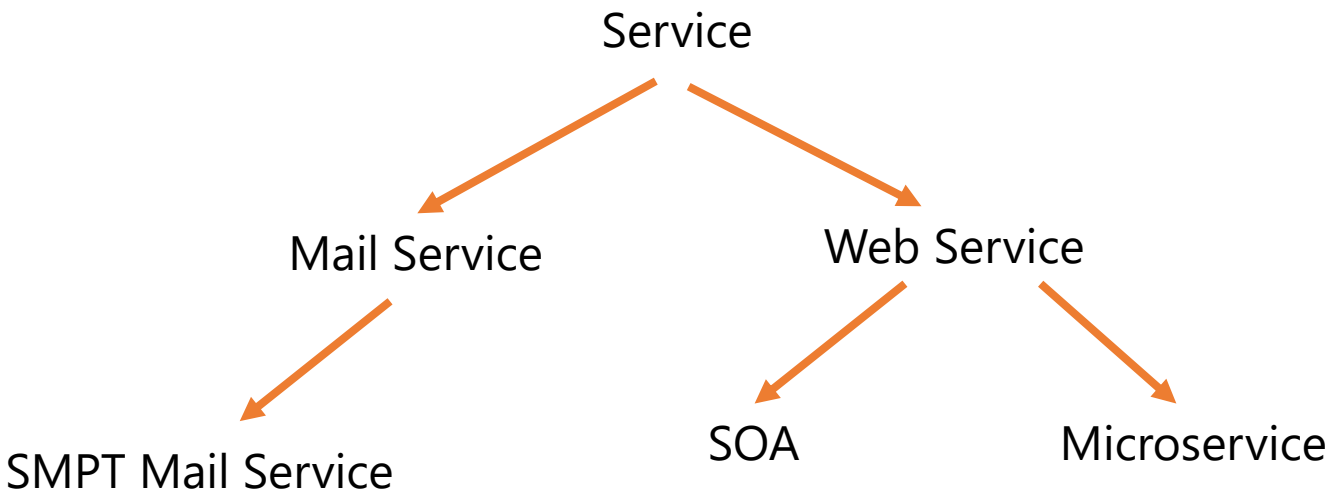
- Abhängigkeiten
  - Controller zu View
  - Controller zu Model
  - View zu Model
- Controller bindet die das Model an die View.

Lab

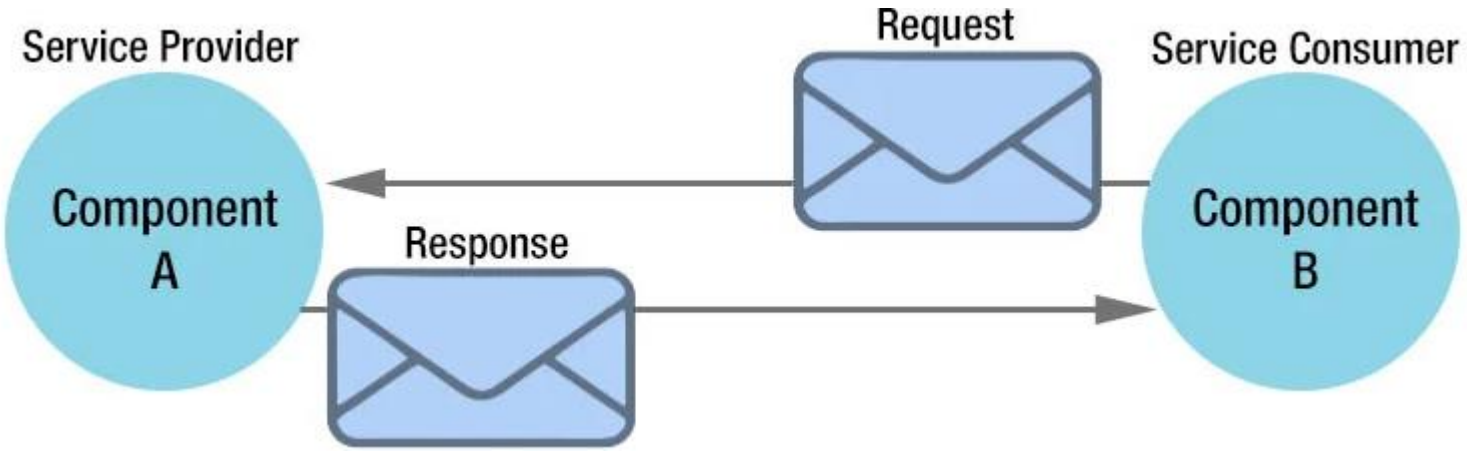
Lab c-07

Standardarchitekturen
Service-orientierte Architektur

- Die Architektur stellt Dienste bereit.
- Hier softwaretechnischer Fokus

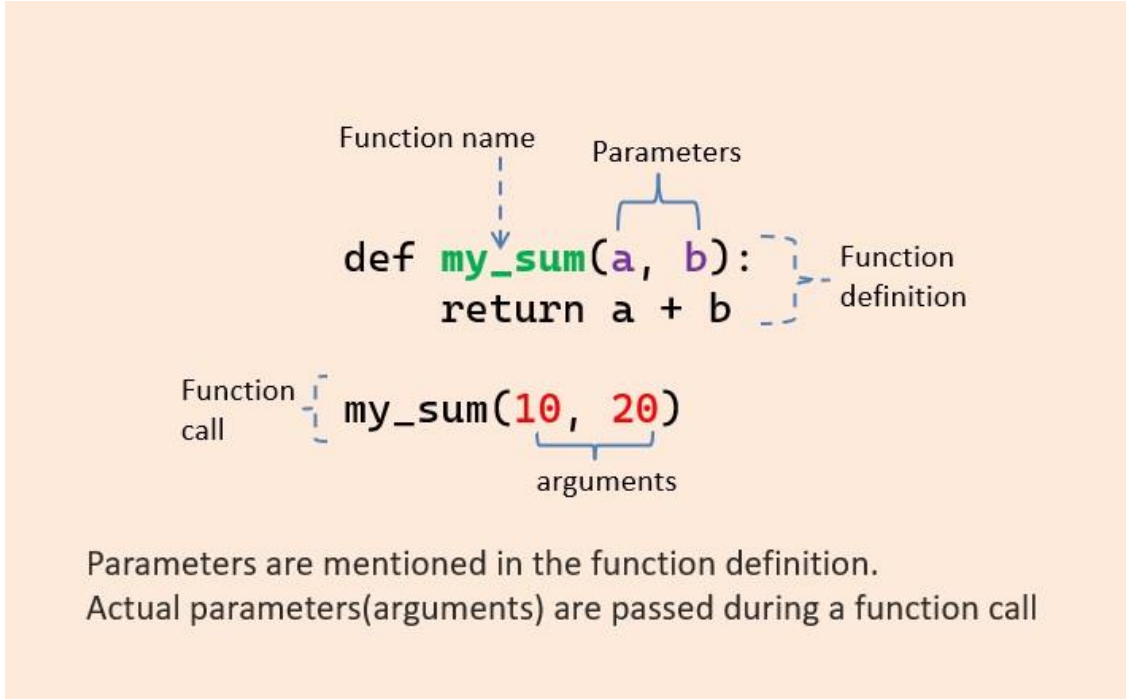


Standardarchitekturen
Service-orientierte Architektur

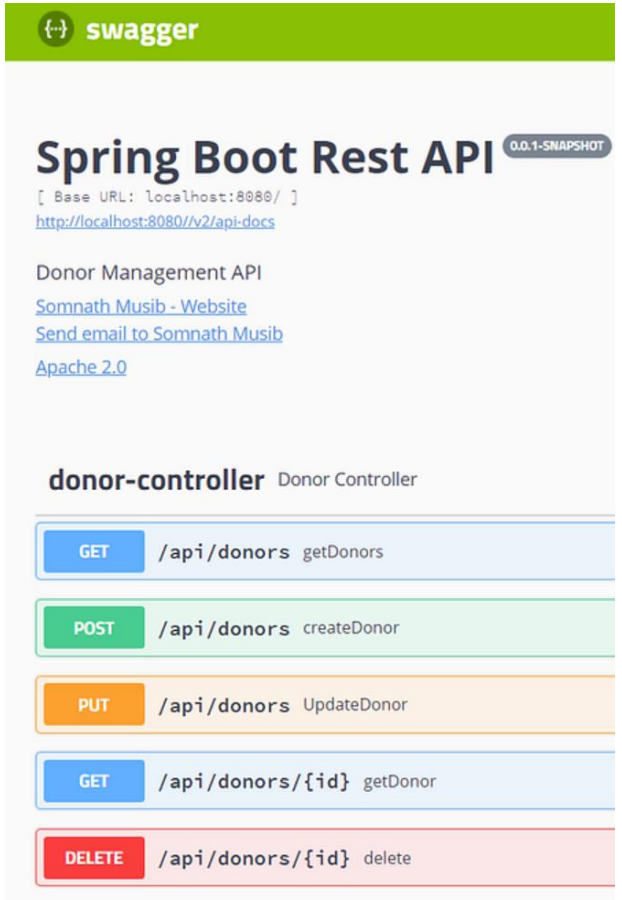


- **Serviceprovider** stellt die „Dienstleistung“ zur Verfügung.
- Der **Servicekonsument** kann die Dienstleistung abrufen/konsumieren.
- Zwischen Provider und Konsument gilt der **Service Contact**/Vertrag.
- Der **Servicekonsument** muss die Dienstleistung **aktiv anfragen** (requesten).

Standardarchitekturen
Service-orientierte Architektur



vs.



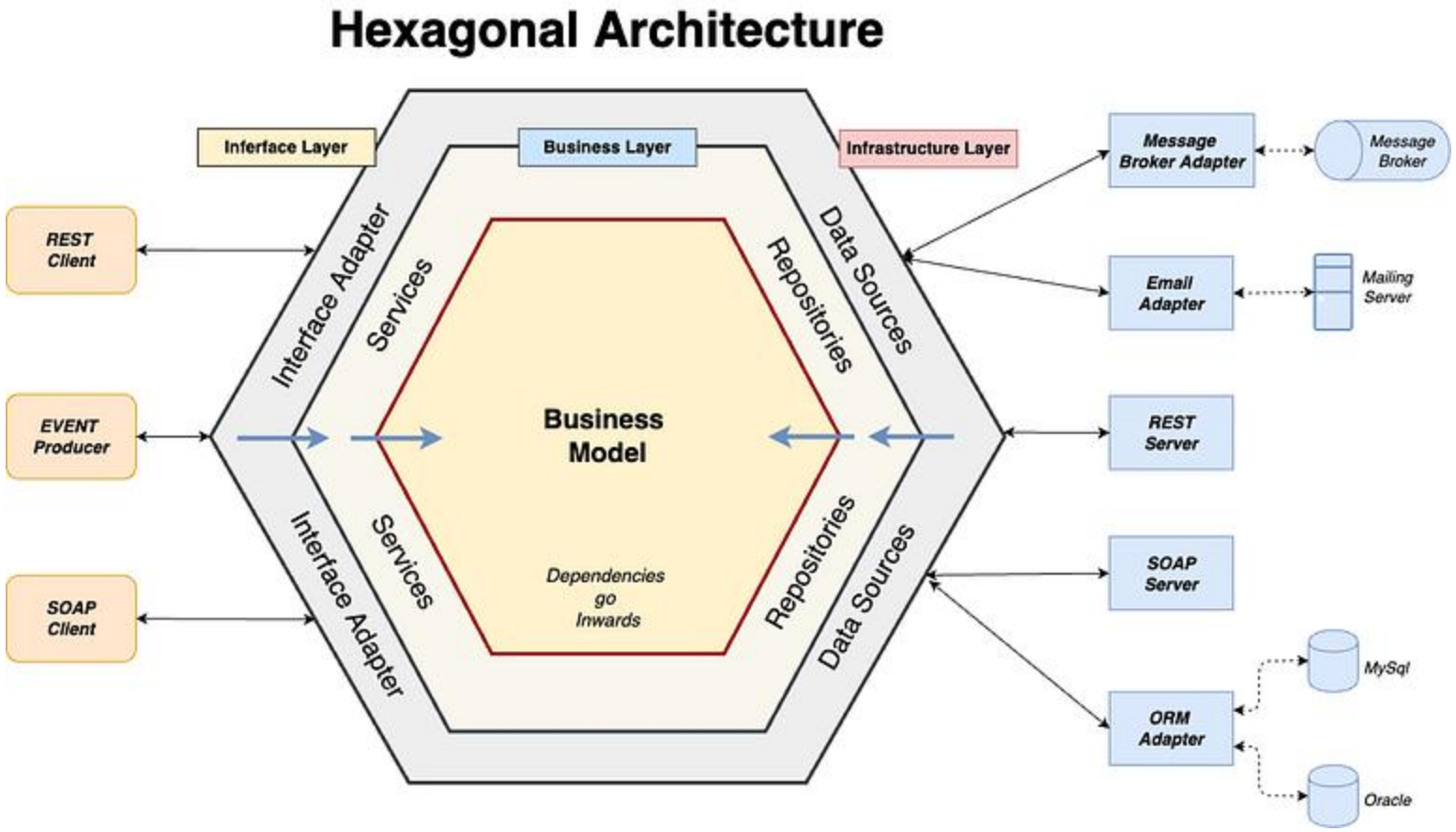
- **Service Contract:**

  Input- vs. Ouput-Parameter
  Verhalten des Dienstes
  Benennung des Dienstes
  Mögliche Fehler des Dienstes

Lab

Lab c-08

Standardarchitekturen
Hexagonale Architektur



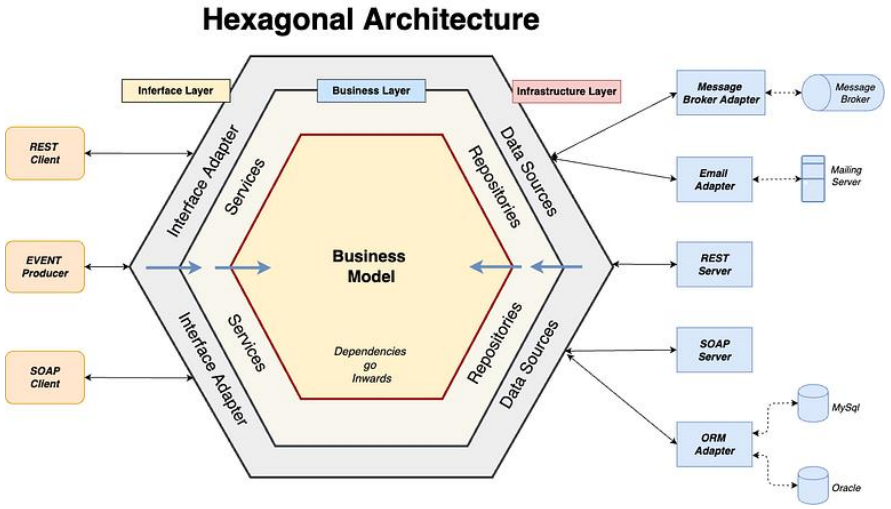
- Angeordnet als Hexagon (Sechseck).
- Port- und adapterbasierte Architektur

Standardarchitekturen
Hexagonale Architektur

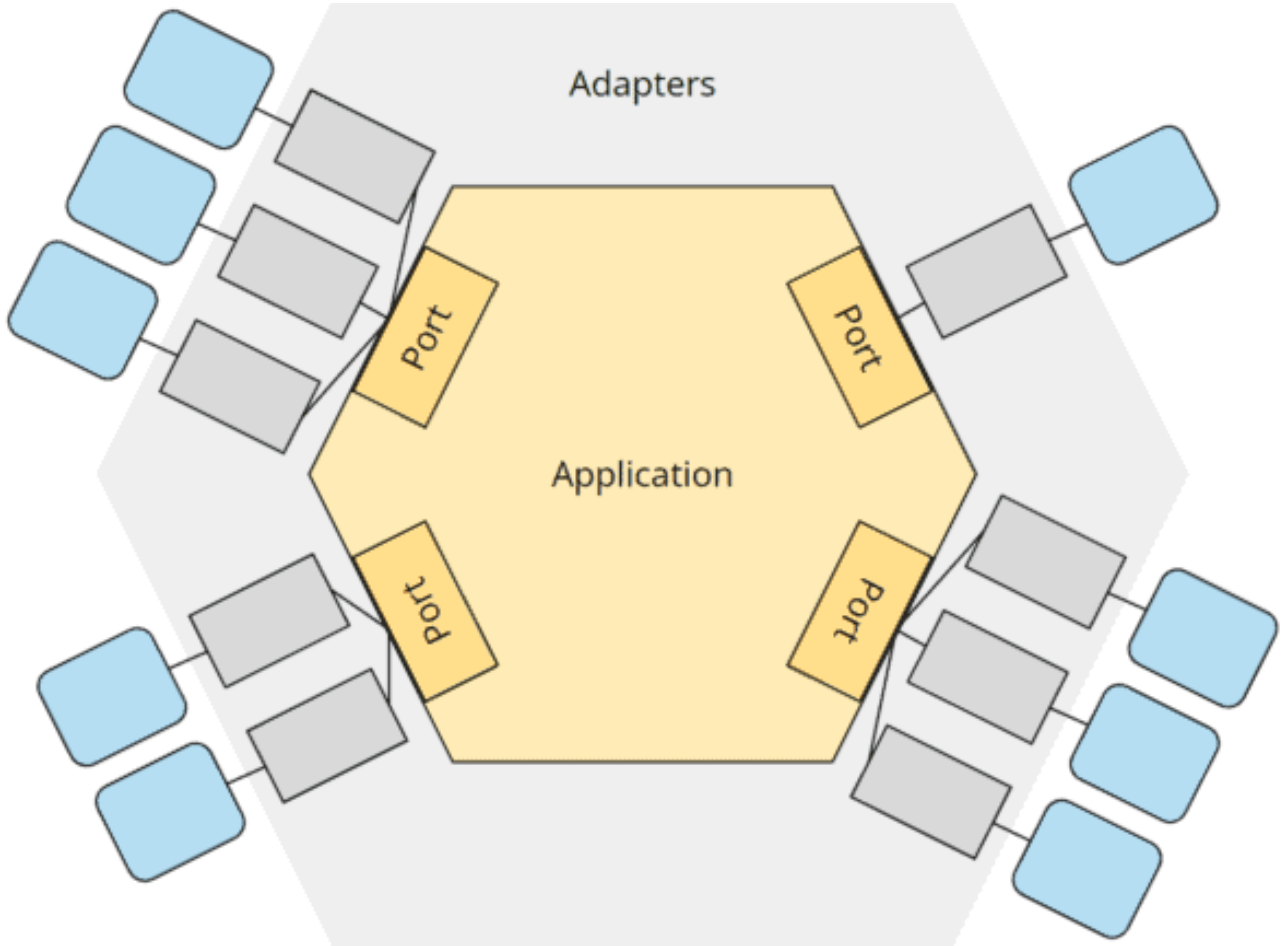
- Entkoppelt die Businesslogik von den Abhängigkeiten

  und technischen Details.

- Drei Hauptelemente
  - **Domain Model**: Enthält Businesslogik und das dazugehörige Datenmodell.
  - **Ports**: Schnittstelle zwischen externer Welt und der Fachlogik.
  - **Adapter**: Transformationskomponenten zwischen externer Welt und den Ports.

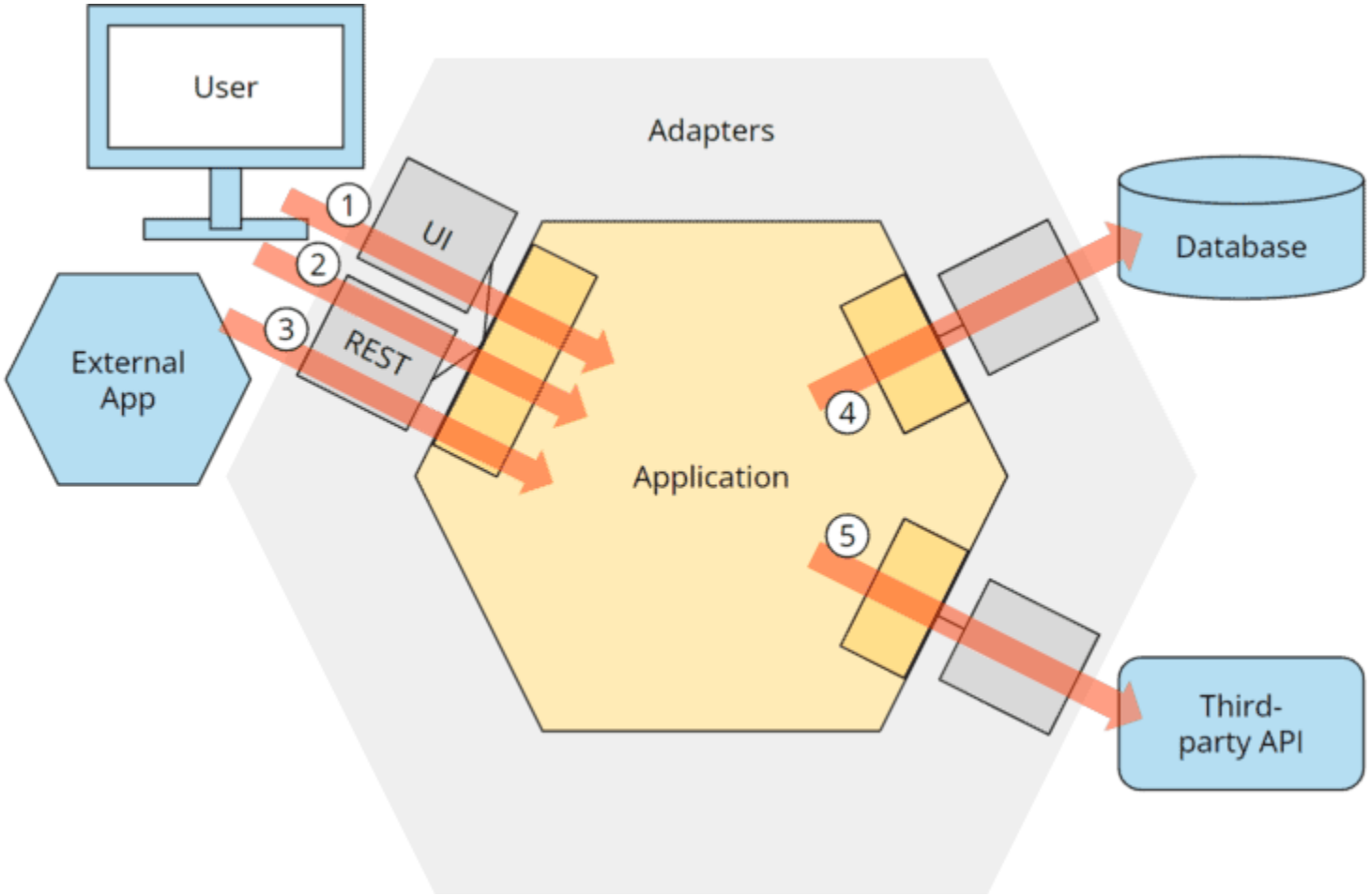


Standardarchitekturen
Hexagonale Architektur



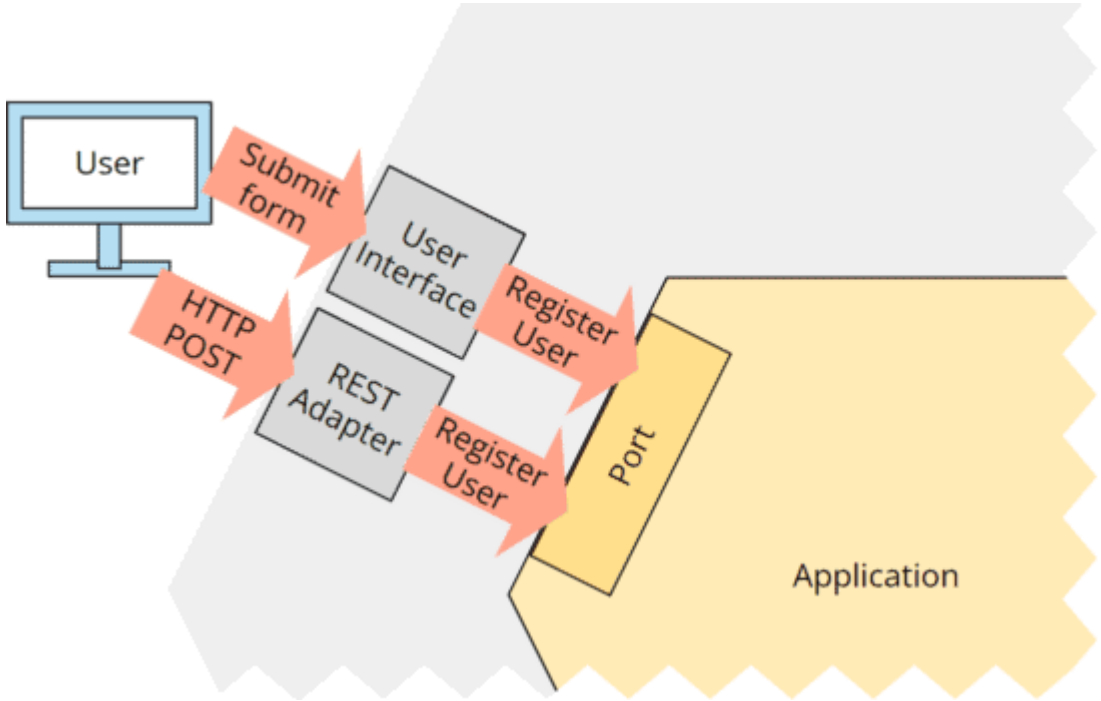
- Allgemeines Beispiel zur Entkopplung der Businesslogik/Application Logic

Standardarchitekturen
Hexagonale Architektur



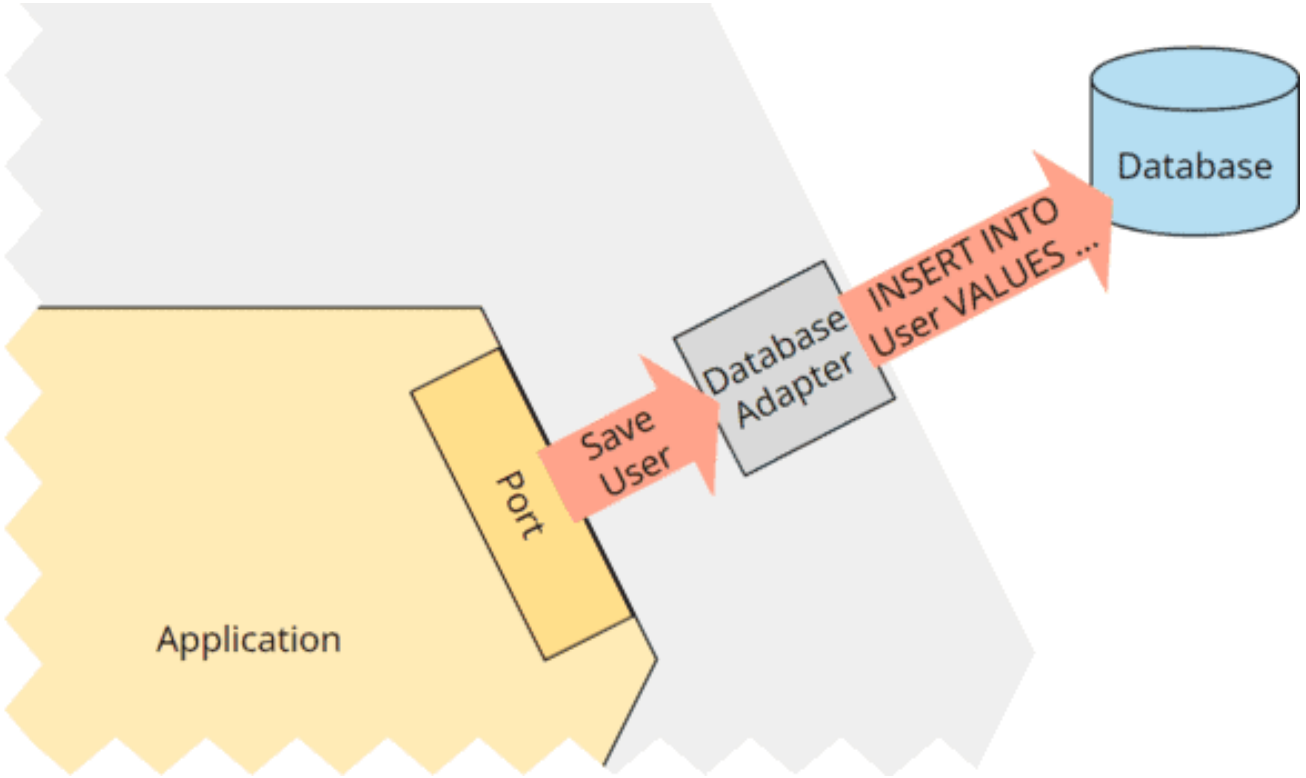
- Allgemeines Beispiel zur Entkopplung der Businesslogik/Application Logic

Standardarchitekturen
Hexagonale Architektur



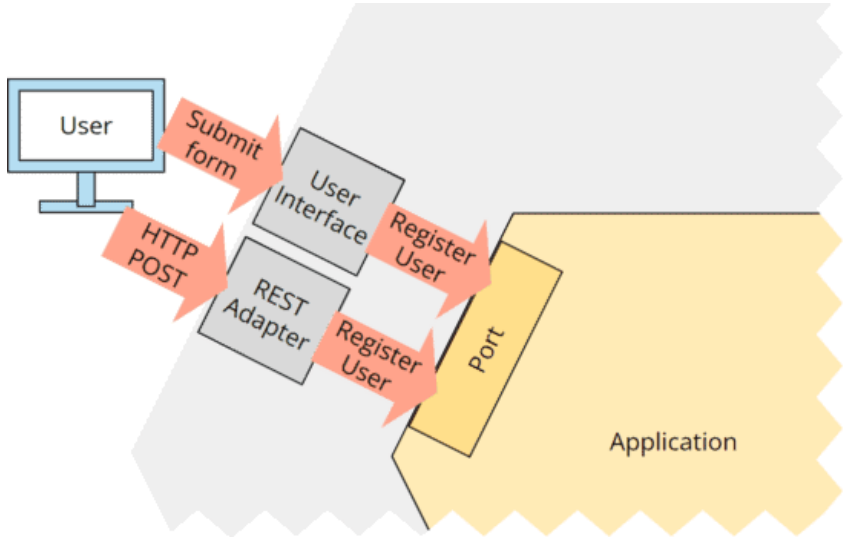
- Beispiel Schnittstellen für ein „Reservierungsformular“.
- **Adapter**:
  - User Interface
  - Datenübergabe/Ermittlung über REST

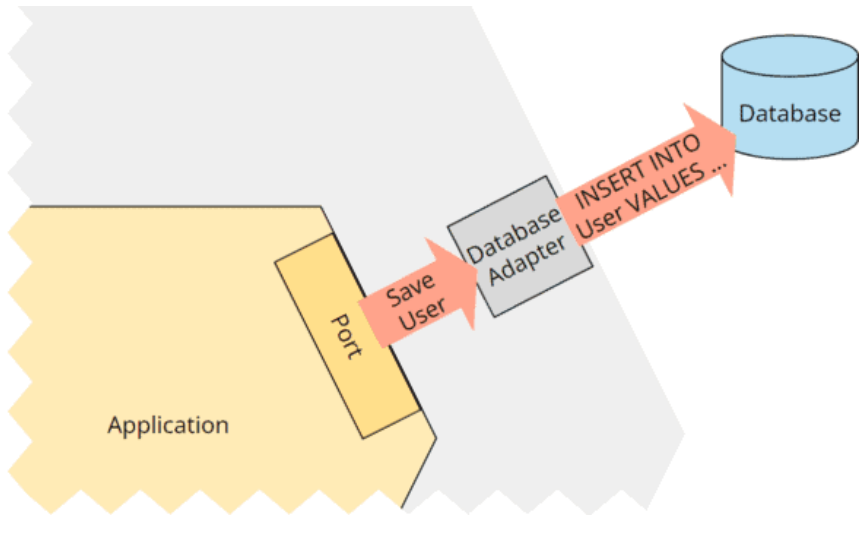
Standardarchitekturen
Hexagonale Architektur



- Dazugehöriger Port, um anschließend die Daten des Reservierungsformulars zu speichern.
- Adapter übersetzt in diesem Fall von „interner Logik“ wie Methodenaufruf zu SQL.

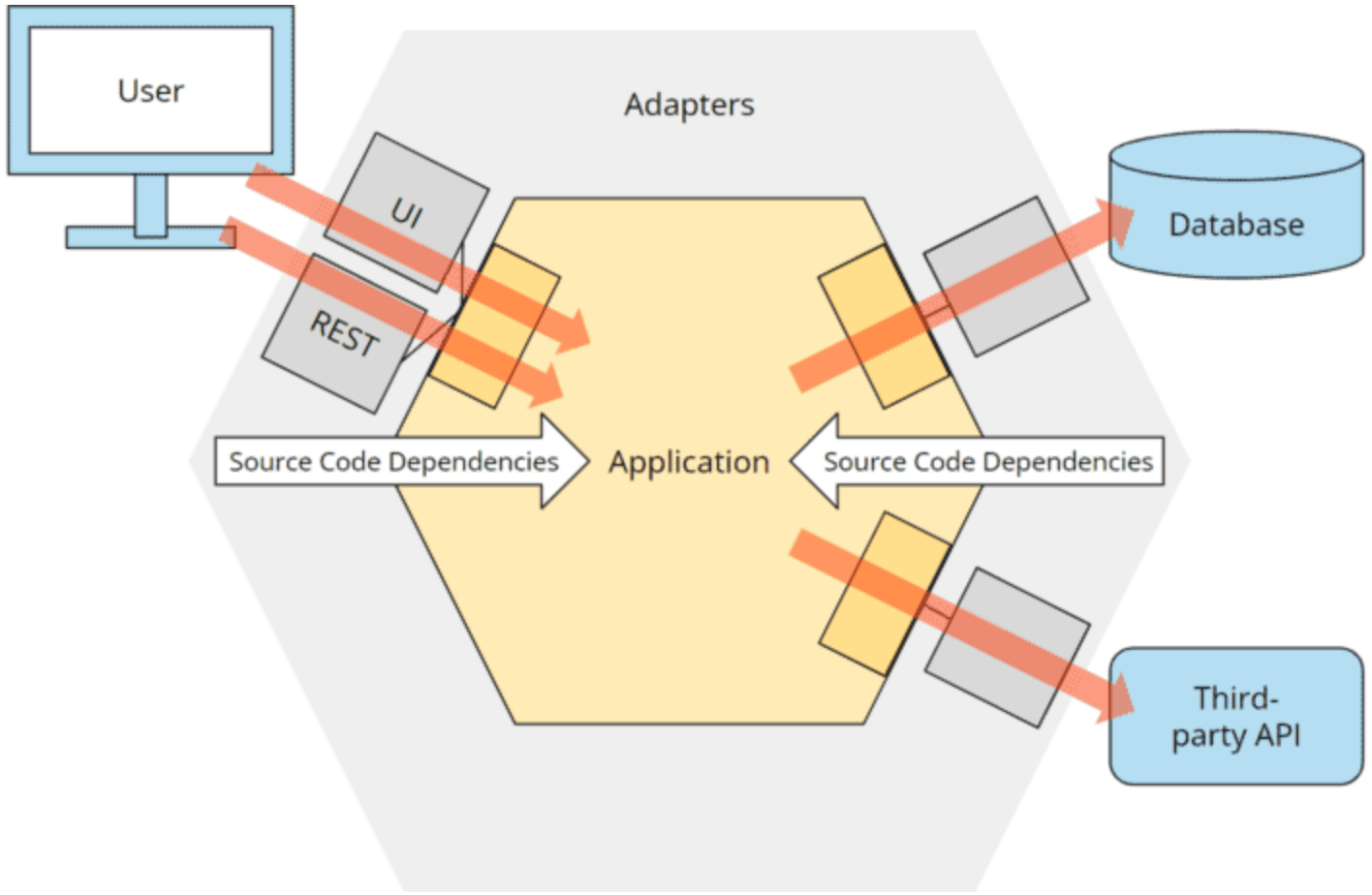
Standardarchitekturen
Hexagonale Architektur





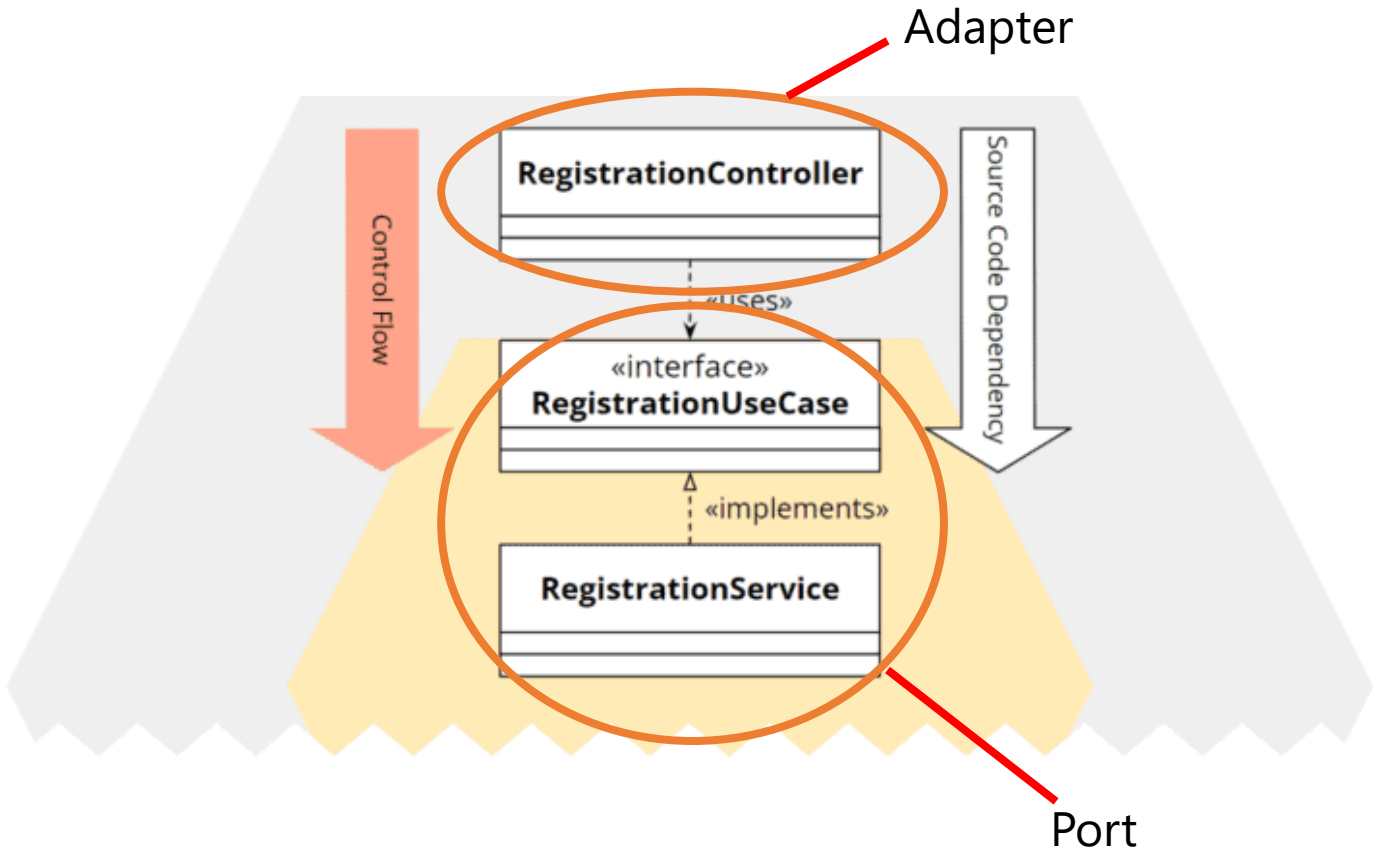
- Es existieren zwei Arten von Adapter:
- **Primäre Adapter**:
  - „steuern“ die Anwendung (Beispiel Formular ausfüllen und abschicken, steuert die Anwendung)
  - Darstellung auf der linken Seite des Hexagons
- **Sekundäre Adapter**:
  - Werden von der Anwendung gesteuert.
  - Darstellung auf der rechten Seite des Hexagons (Bsp. Datenbankaufruf).

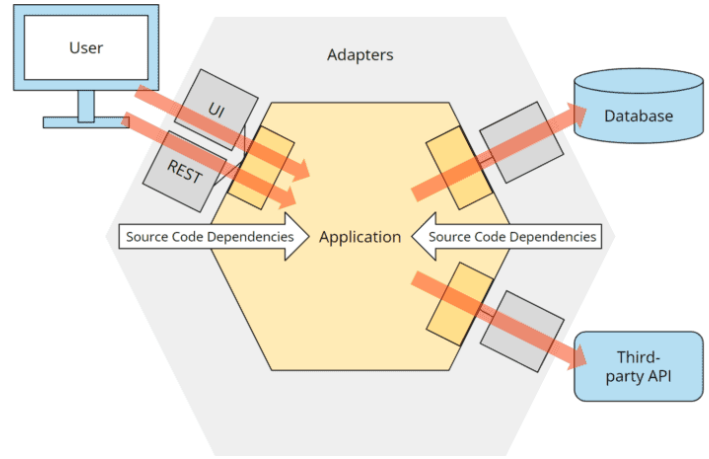
Standardarchitekturen
Hexagonale Architektur



- Dependency Rule zur praktischen Umsetzung:
  - Abhängigkeiten im Source Code nur von außen nach innen zulässig!

Standardarchitekturen
Hexagonale Architektur



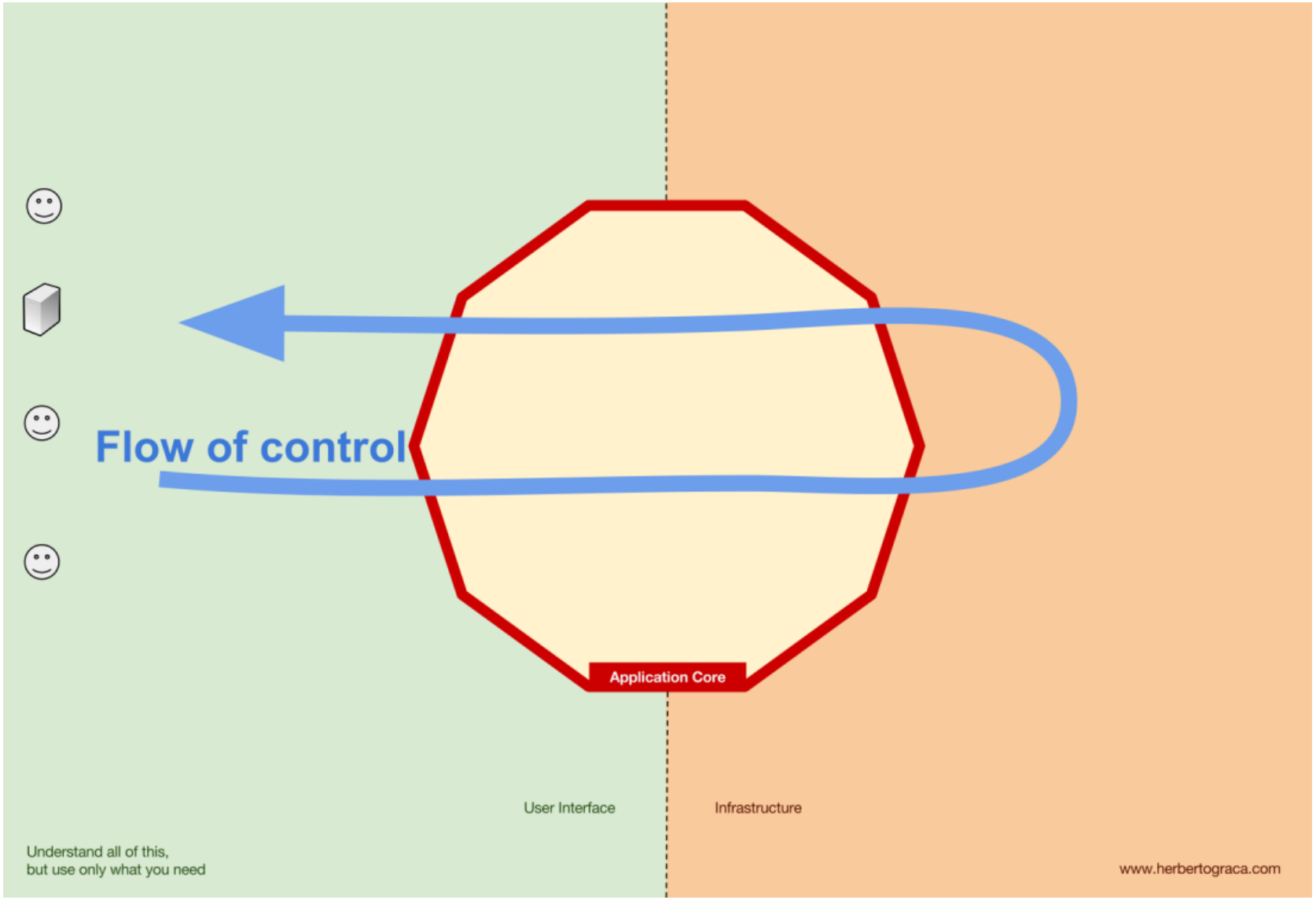


- Dependency Rule zur praktischen Umsetzung:
  - Abhängigkeiten im Source Code nur von außen nach innen zulässig!

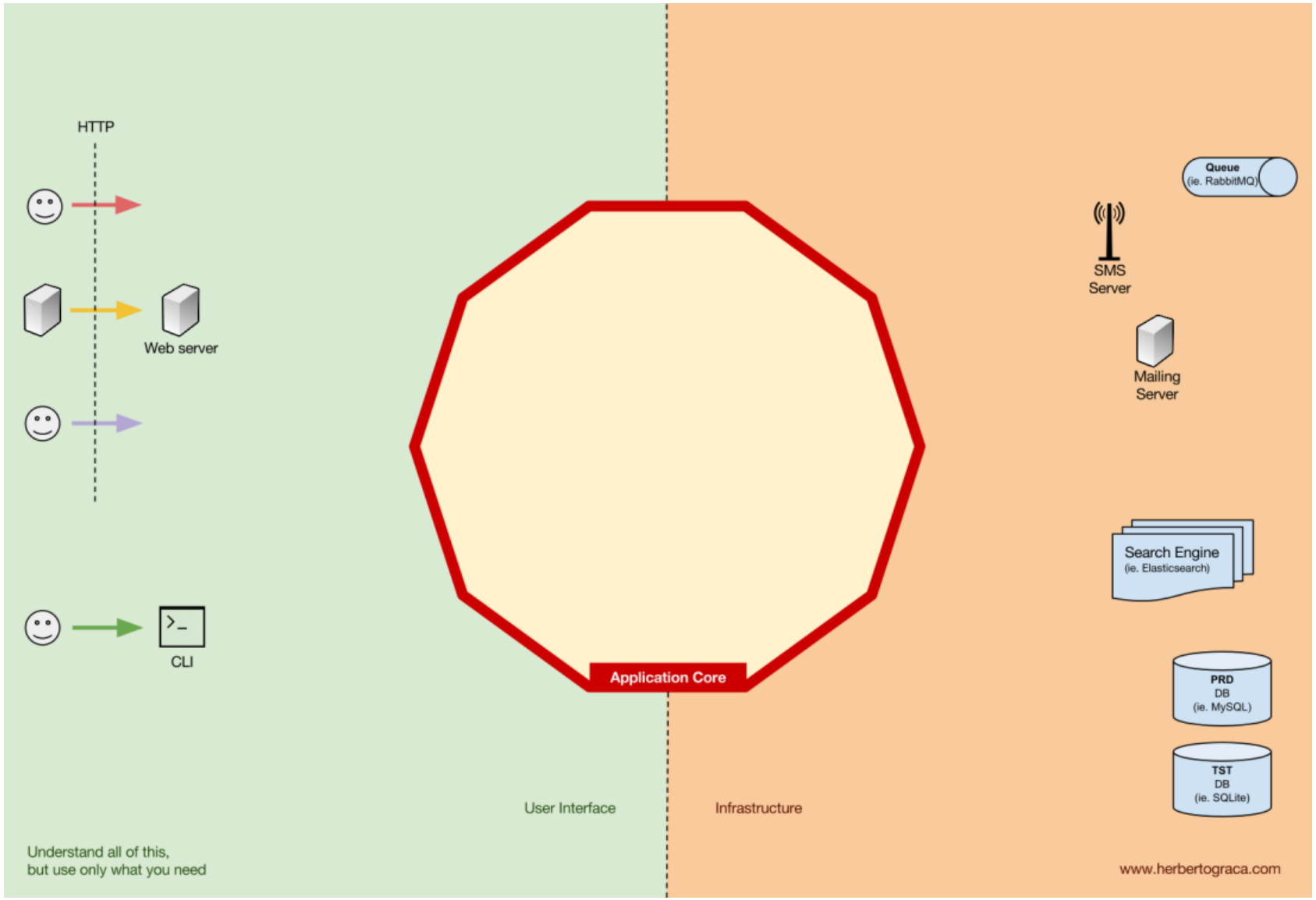
Lab

Lab c-09

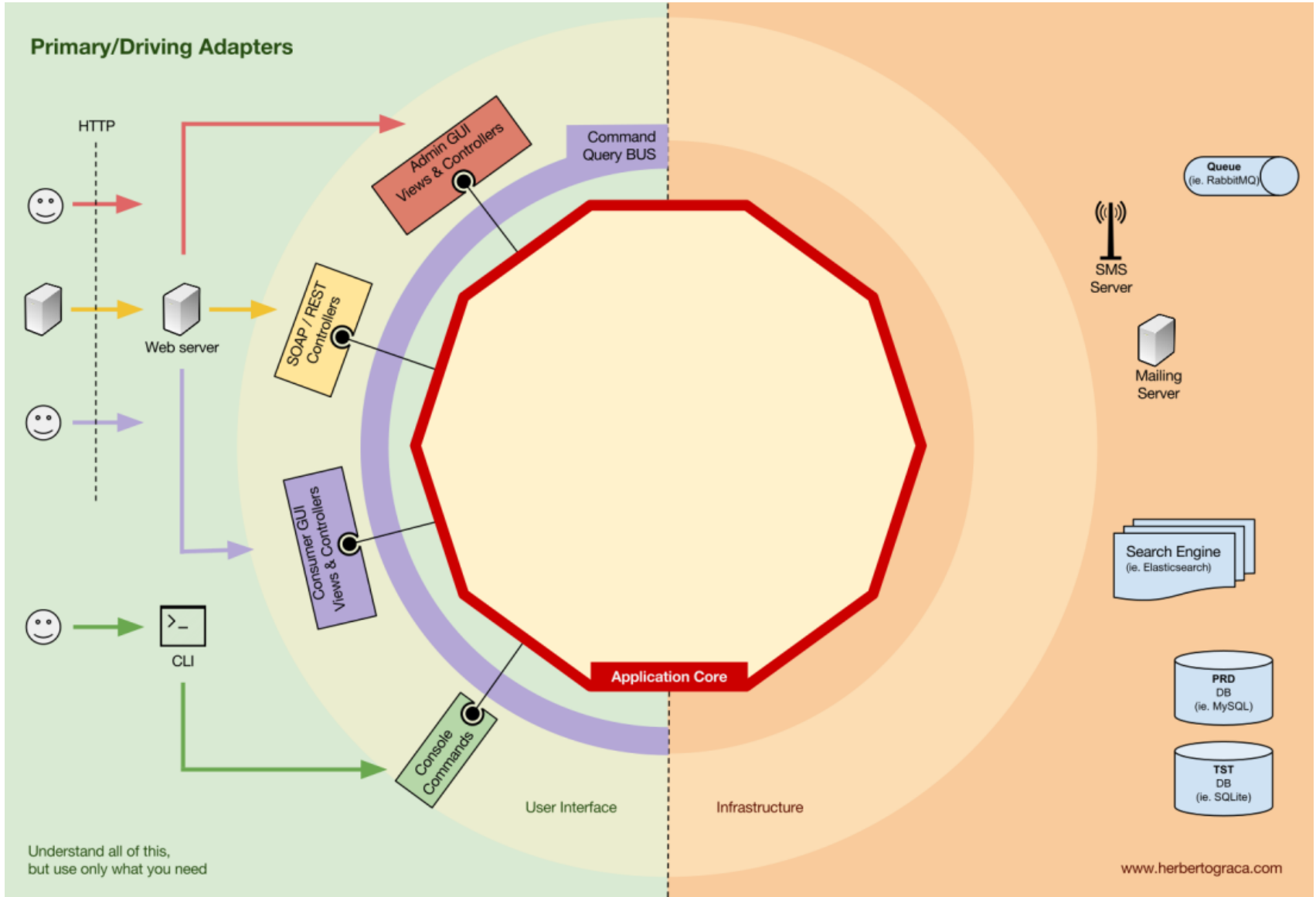
Standardarchitekturen
Hexagonale Architektur Beispiel



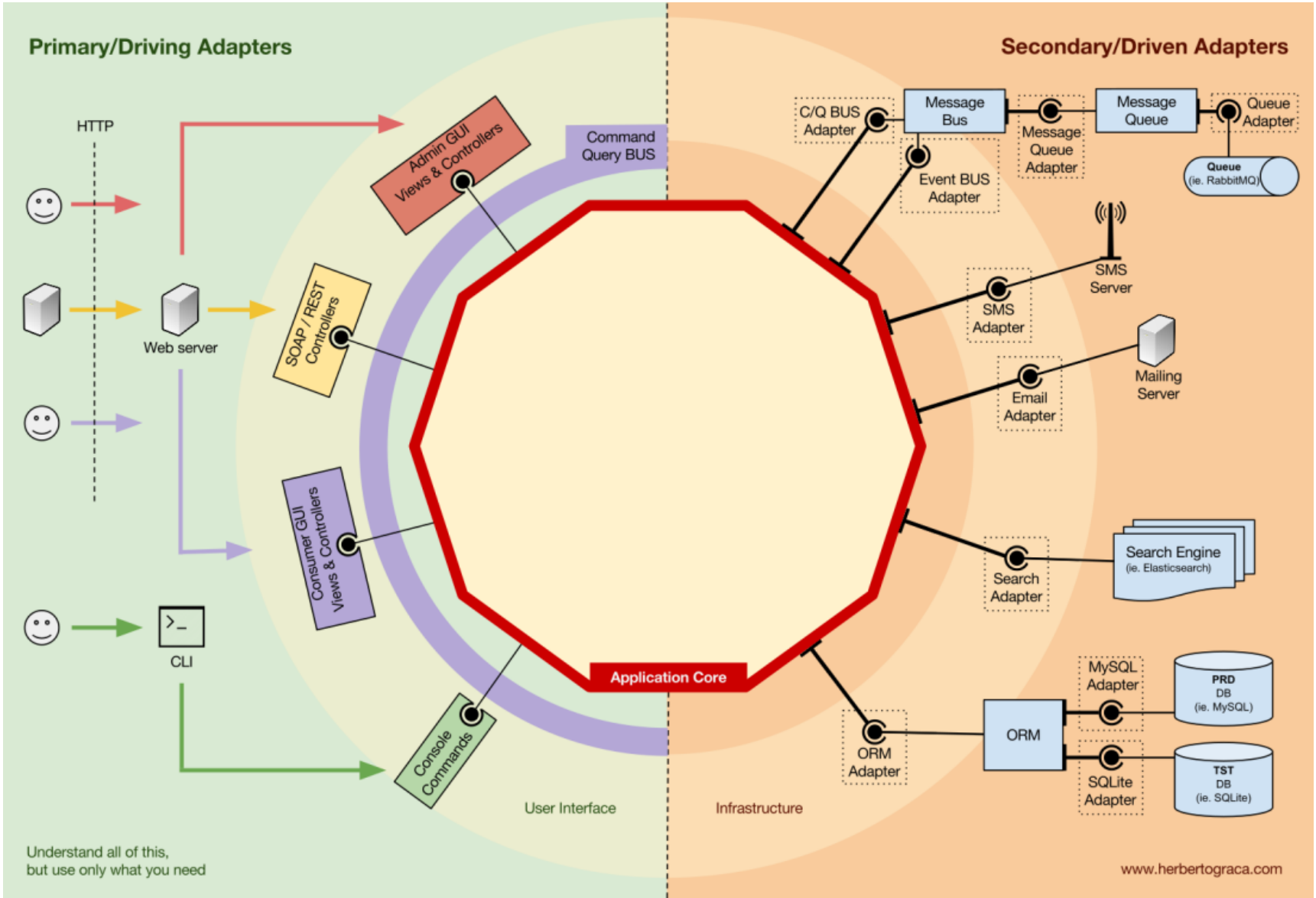
Standardarchitekturen
Hexagonale Architektur Beispiel



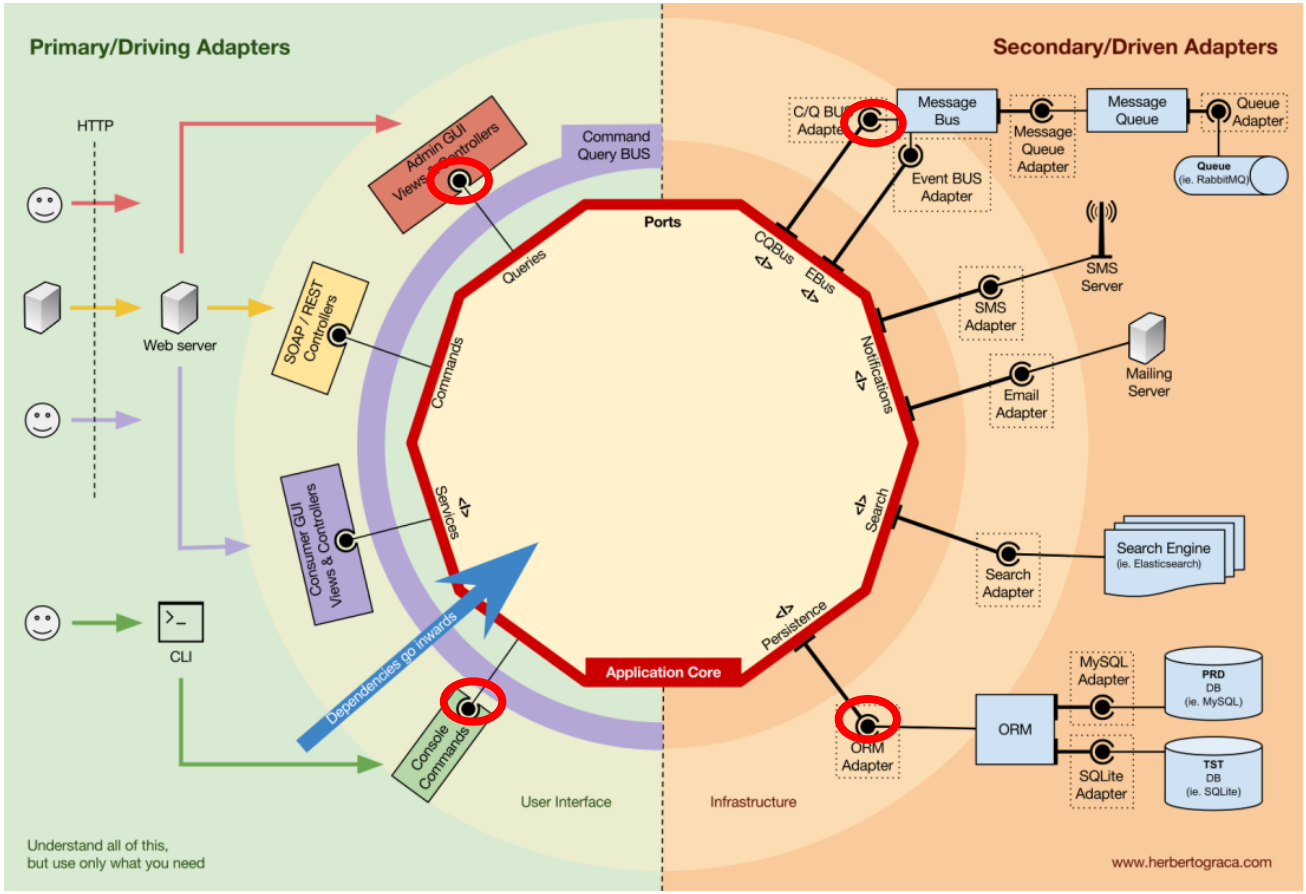
Standardarchitekturen
Hexagonale Architektur Beispiel



Standardarchitekturen
Hexagonale Architektur Beispiel

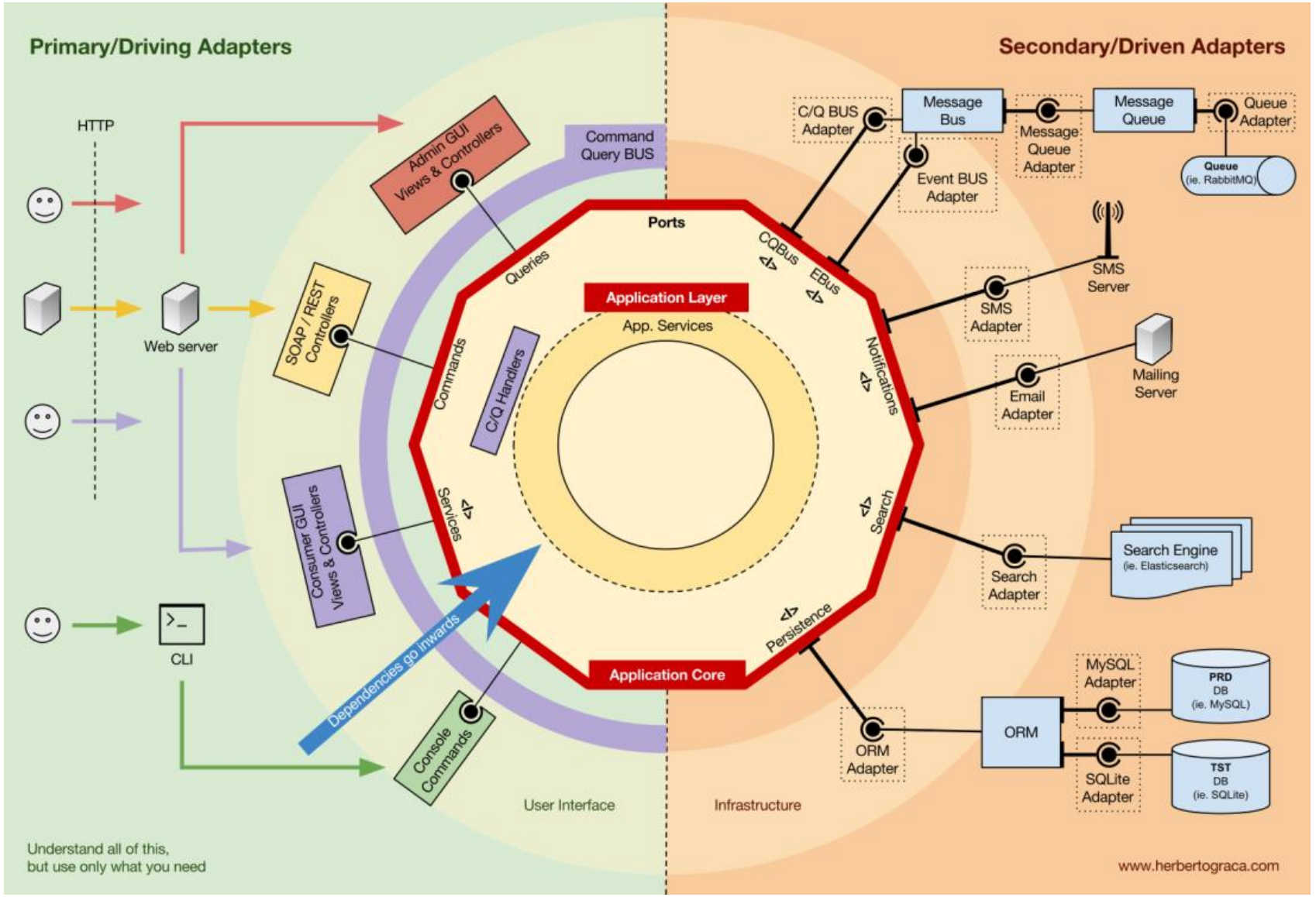


Standardarchitekturen
Hexagonale Architektur Beispiel

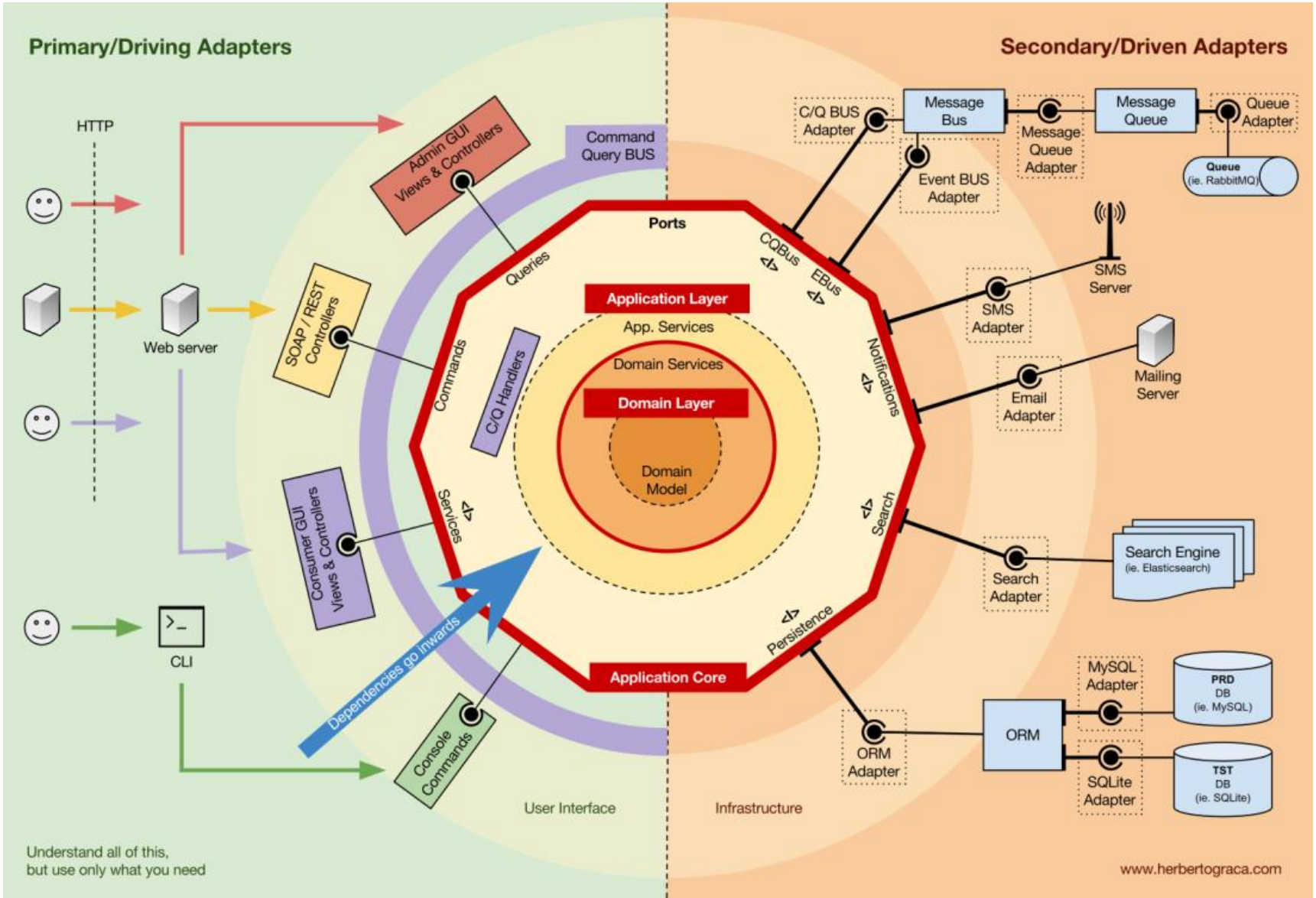


- **Adaptoren** hängen von **specifizschen tools** und einem spezifischen **port** ab**.**
- Businesslogik hängt hingegen nur vom **port** (interface) ab.

Standardarchitekturen
Hexagonale Architektur Beispiel



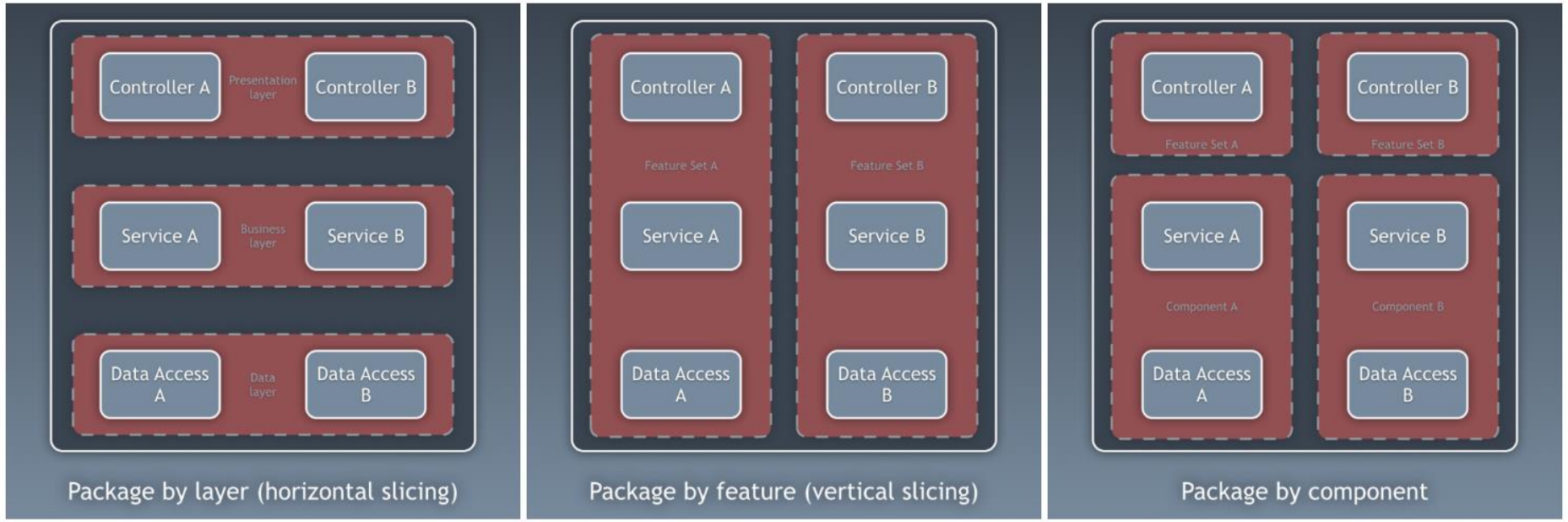
Standardarchitekturen
Hexagonale Architektur Beispiel



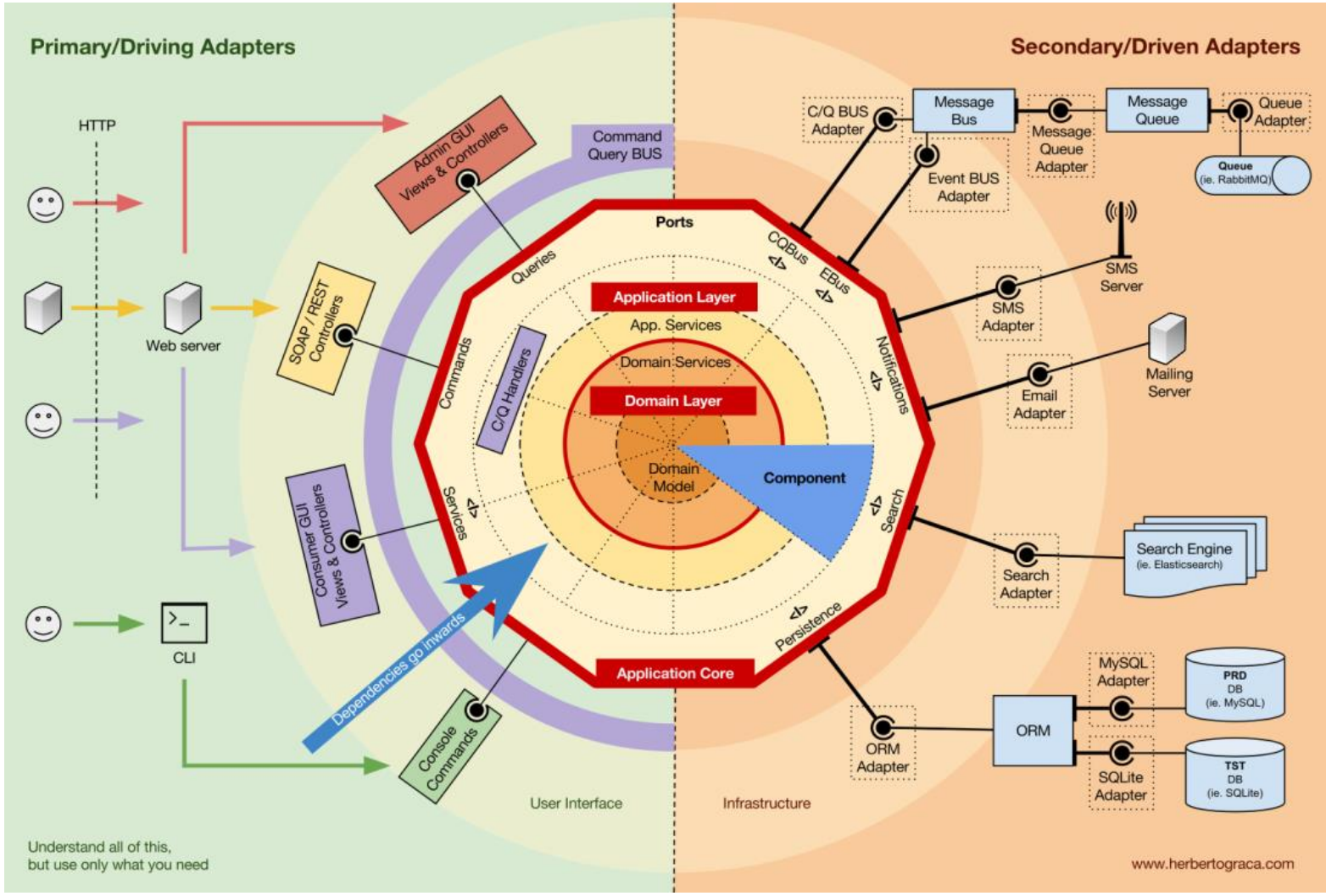
Standardarchitekturen
Hexagonale Architektur Beispiel

- Bisher haben wir den Programmcode in Layer organisiert.
- Das ist eine sehr feingranulare Trennung.
- Im Normalfall wird auf Komponentenebene getrennt.

- Komponenten können nach Layer, Feature oder Businesskontext getrennt werden.

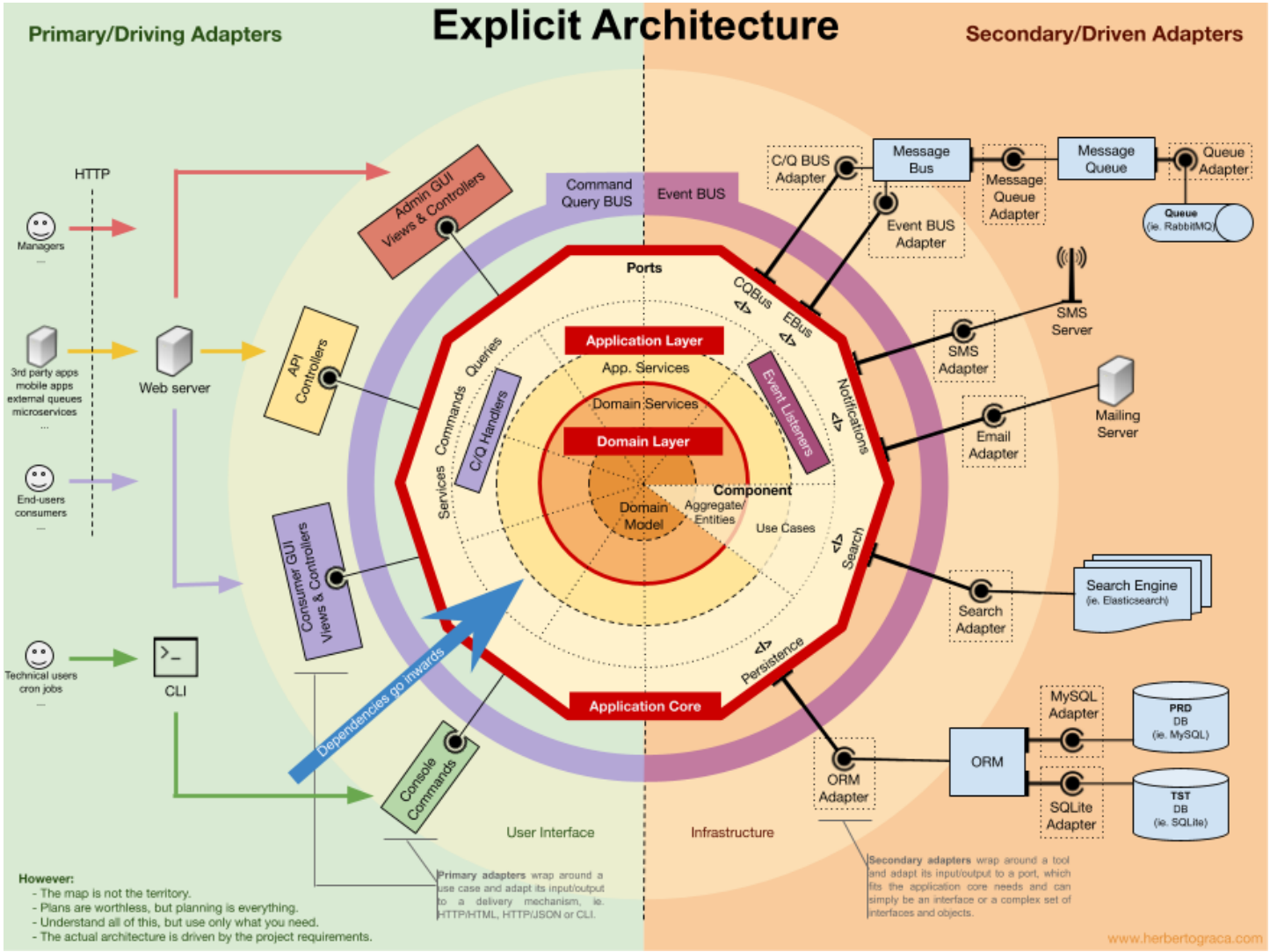


Standardarchitekturen
Hexagonale Architektur Beispiel



- Komponente überspannt mehrere Schichten.

Standardarchitekturen
Hexagonale Architektur Beispiel



- Entkopplung der Komponentenkommunikation durch **asynchrone** Kommunikation.

Komponentenbasierte Softwareentwicklung

- Setzt auf Wiederverwendung von Softwareeinheiten, den sogenannten „Softwarekomponenten“.

- Komponenten sind abstrakter und können als eigenständige Einheiten betrachtet werden.

- Jede Komponente sollte einen fachlichen Bereich durch Fachfunktionen abbilden.
- Die Trennung in Komponenten erfolgt demzufolge basierend auf der Geschäftslogik, nicht auf Grund technischer Gegebenheiten.

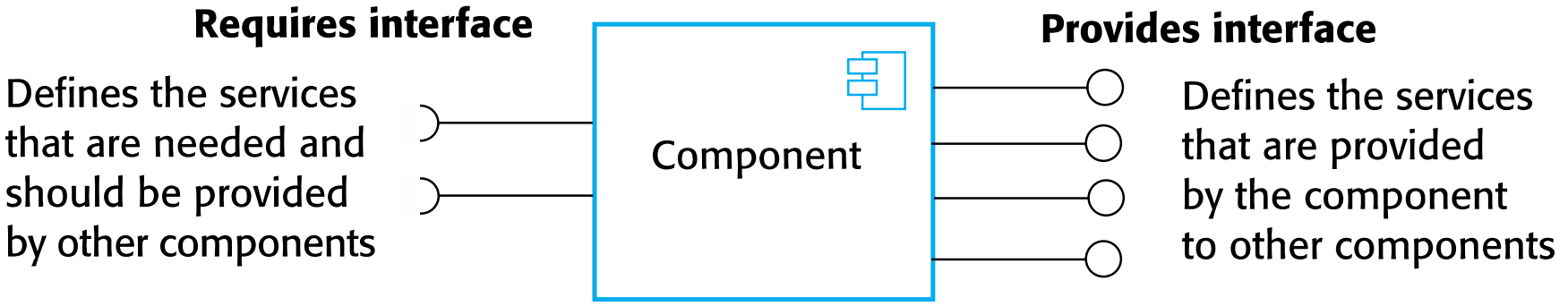
Komponentenbasierte Softwareentwicklung

- Wichtige Designprinzipien
  - Wiederverwendbarkeit
  - Unabhängigkeit
  - Autonome Einheit
  - Implementierung der Komponenten ist verborgen
  - Kommunikation erfolgt über wohldefinierte Schnittstellen
  - Komponente kann ersetzt werden, wenn ihre Schnittstelle beibehalten wird
- Service-Orientierung ist eine spezielle Ausprägung einer Komponente
- Service-Orientierung != SOA
- Service-Orientierung wird heute vielfach eingesetzt

Komponentenbasierte Softwareentwicklung

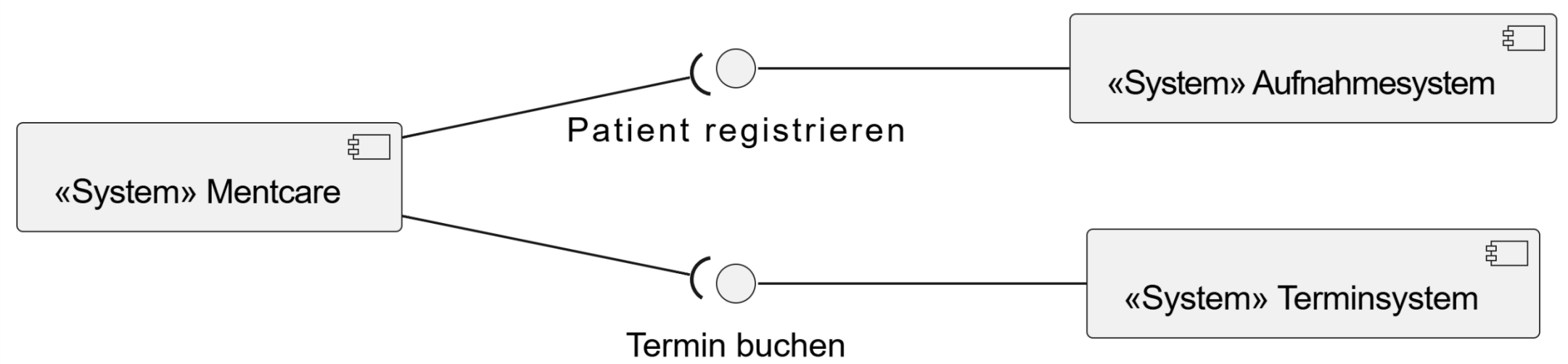
- Wichtige Eigenschaften einer Softwarekomponente
  - Composable (Zusammensetzbar)
  - Deployable (Einsatzfähig)
  - Documented (Beschrieben, Dokumentiert)
  - Independent (Unabhängig)
  - Standardized (Standardisiert)
- Unterscheidung zwischen anbietender und benötigter Schnittstelle (Interface).
- **Anbietende Schnittstelle**
  - Definiert die Dienste, die für andere bereitgestellt werden (Komponenten-API).
- **Benötigte Schnittstelle** –
  - Definiert die Dienste, damit die Komponente wie angegeben ausgeführt werden kann.
  - Beeinträchtigt nicht Unabhängigkeitskriterium.
  - „benötigt" definiert nicht „wie“ die Dienste bereitgestellt werden sollen.

Komponentenbasierte Softwareentwicklung



- UML Komponentendiagramm
- Gibt die „**anbietende**“ (rechts) und die „**benötigte**“ (links) Schnittstellennotation an.

Komponentenbasierte Softwareentwicklung



- Beispielkomponente aus dem Mentcare System.
- Eine Komponente kann auch abstrakter modelliert werden, so dass nicht alle Fachfunktionen detailliert dargestellt werden müssen.

Komponentenbasierte Softwareentwicklung
Komposition

- **Komposition**: Prozess des **Zusammenfügens** von **Komponenten** zu einem **System**.

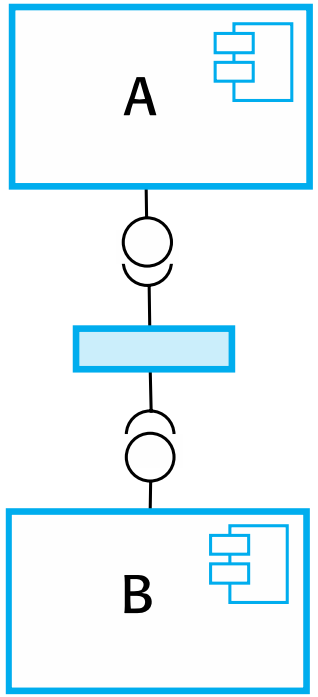
- Die Komposition beinhaltet die Integration von Komponenten untereinander sowie mit der Komponenteninfrastruktur.
- Normalerweise müssen Sie "**Glue Code**" schreiben, um Komponenten zu integrieren.
- **Glue Code**:
  - Programmcode, der es den Komponenten ermöglicht, zusammenzuarbeiten.
  - Glue-Code kann verwendet werden, um Schnittstelleninkompatibilitäten aufzulösen.

Komponentenbasierte Softwareentwicklung - Komposition

- Es wird zwischen drei Kompositionstypen unterschieden

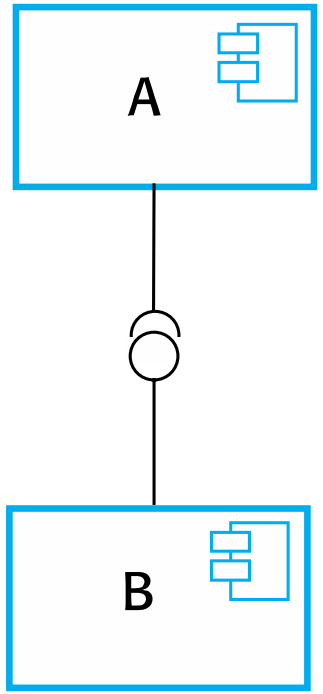
1. **Sequential Composition**
   - Komponenten werden nacheinander (sequenziell) ausgeführt.
   - Es werden die „anbietenden“ Schnittstellen der einzelnen Komponenten zu geeignet zusammengesetzt.
2. **Hierarchical Composition**
   - Komponente nimmt die Dienste einer anderen in Anspruch.
   - „anbietende“ Schnittstelle einer Komponente wird mit der „benötigten“ Schnittstelle der anderen verbunden.
3. **Additive Composition**
   - Schnittstellen zweier Komponenten werden zusammengefügt, um eine neue Komponente zu erzeugen.
   - Die „anbietenden“ und „benötigten“ Schnittstellen der neuen Komponente sind eine Kombination der Schnittstellen der einzelnen enthaltenen Komponenten.

Komponentenbasierte Softwareentwicklung - Komposition



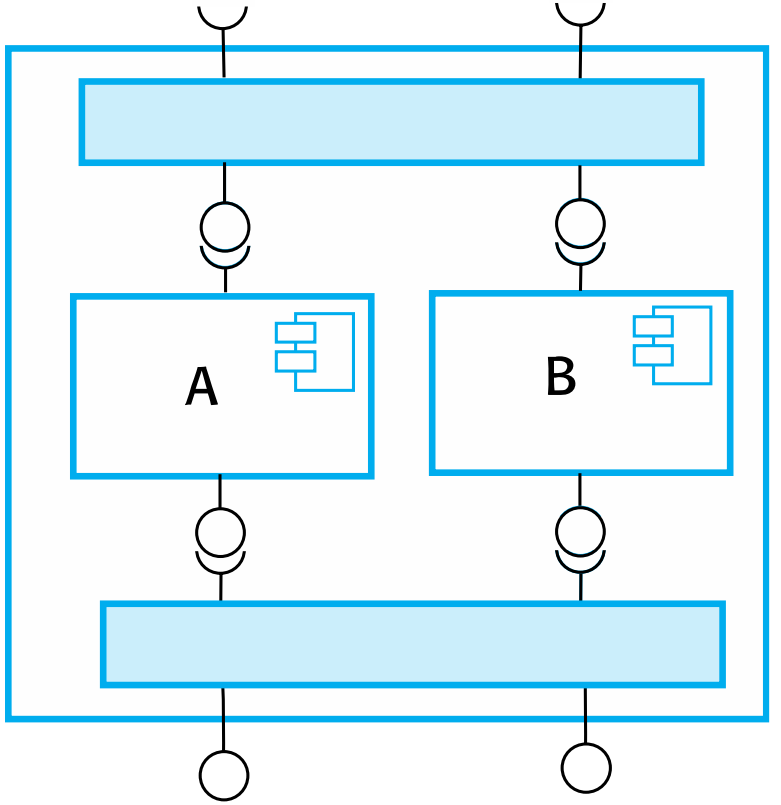
(1)

Sequential Composition



(2)

Hierarchical Composition



(3)

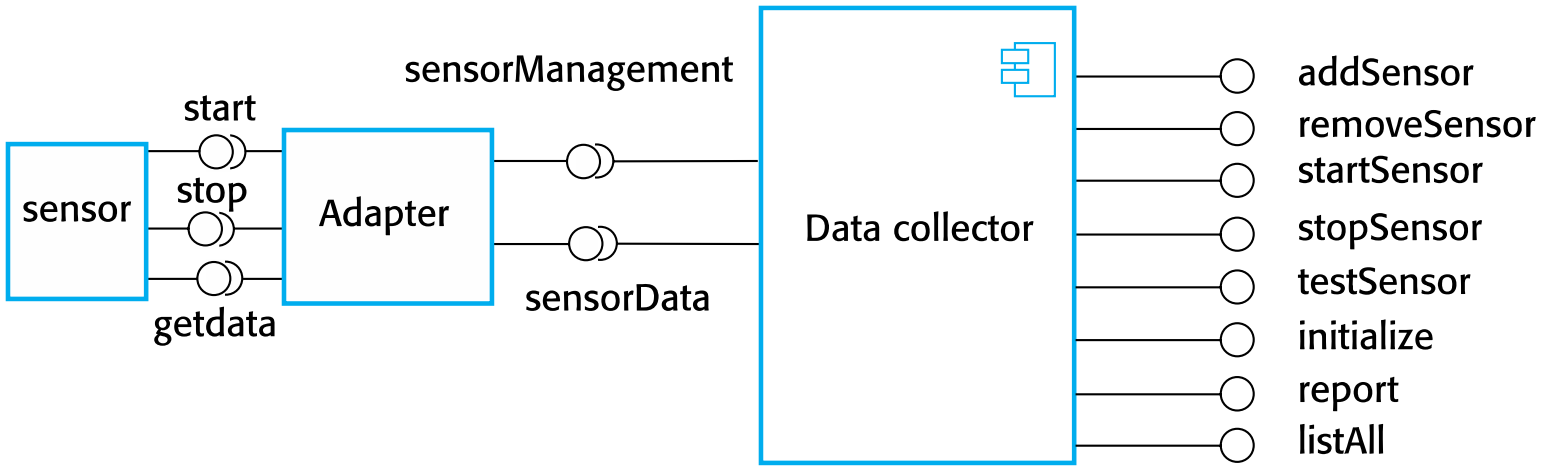
Additive Composition

Komponentenbasierte Softwareentwicklung
Komposition

- Häufig sind die Schnittstellen von Komponenten **inkompatibel zueinander**.
- Um diese dennoch integrieren zu können, muss **Glue Code** erstellt werden.

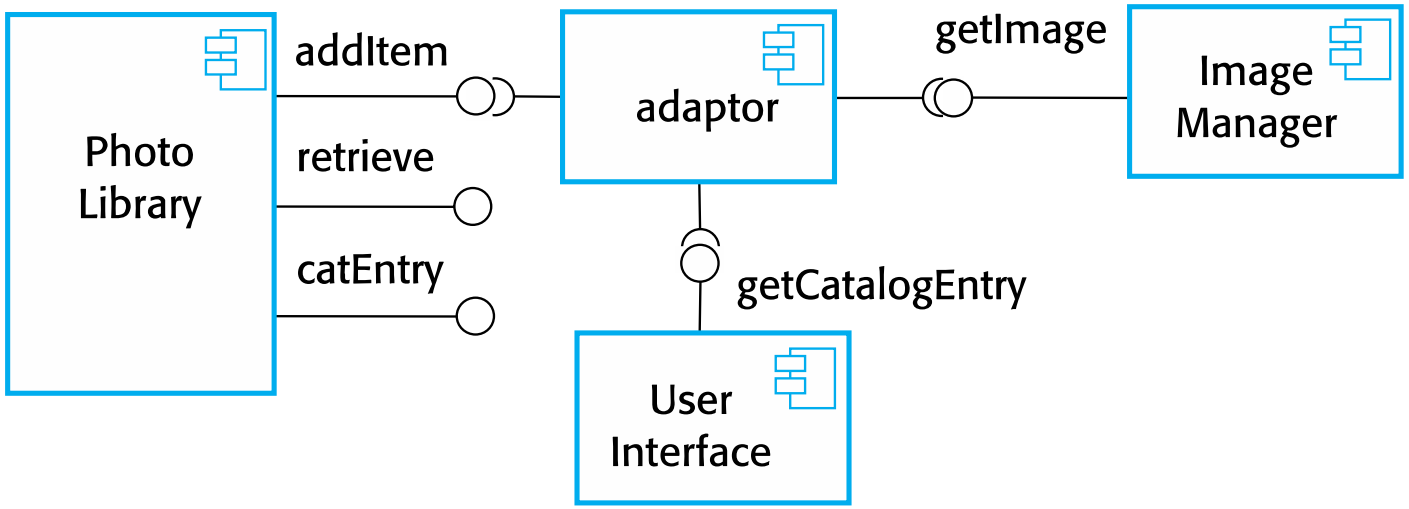
- Glue Code gehört jedoch zu keiner existierenden Komponente.
- Glue Code muss in einer neuen bzw. getrennten „Glue-Code-Komponenten“ eingefügt werden.
- **Glue-Code-Komponente = Adapter.**
- Adapter kann mehr als nur zwei Komponenten zusammenführen.

Komponentenbasierte Softwareentwicklung - Komposition



- Zusammenführung von zwei Komponenten.
- Adapter enthält einfachen **Glue Code** in Form von „**Daten-Mapping-Logik**“.

Komponentenbasierte Softwareentwicklung - Komposition



- Zusammenführung von mehreren Komponenten
- Adapter enthält neben „**Daten-Mapping-Code**“ auch **Aufruflogik** (**Sequenzen**, ...).