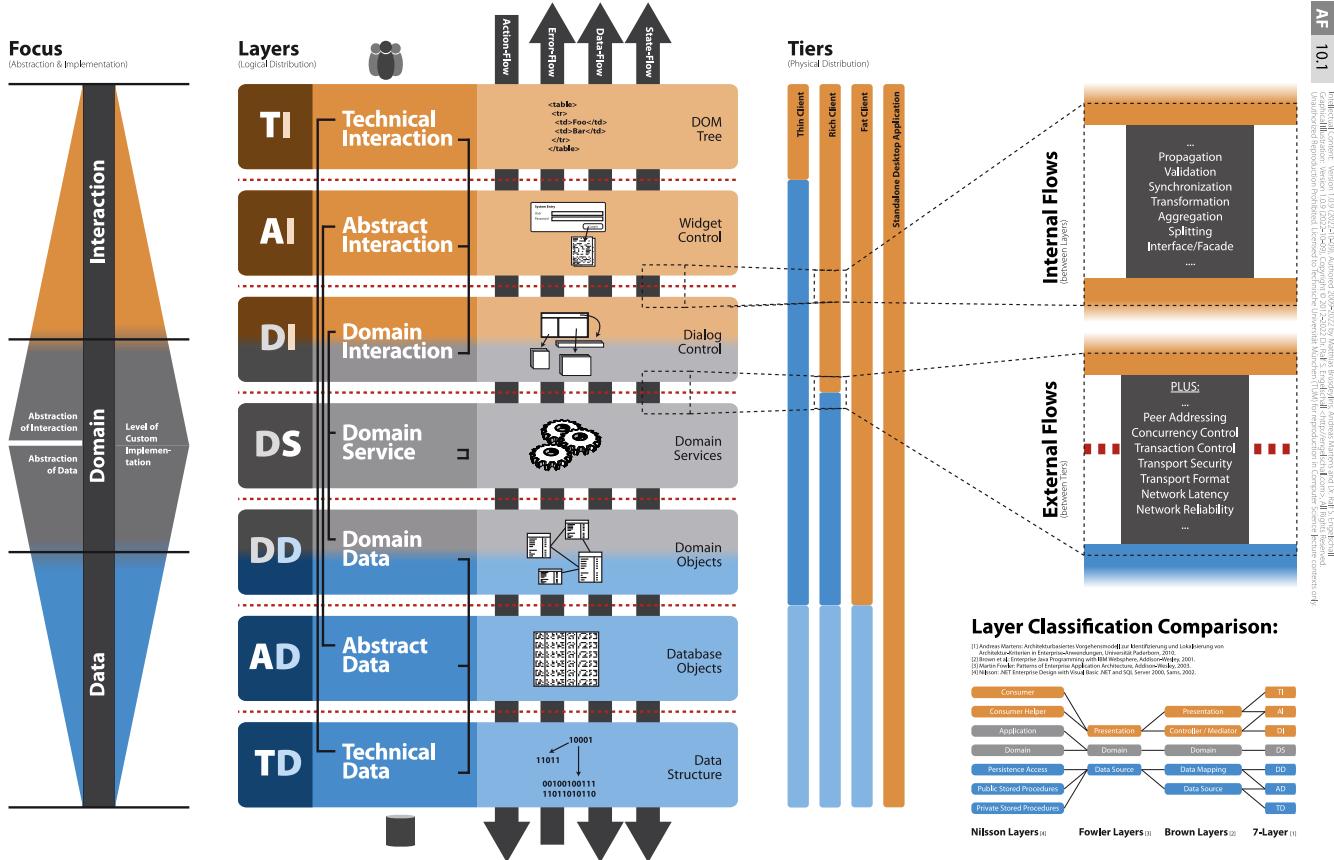




# Software Engineering in der industriellen Praxis (SEIP)

Dr. Ralf S. Engelschall



In einer Anwendung kann man 7 logische Layer unterscheiden, die jeweils auf zwei Arten gruppiert sind: einerseits gibt es die drei sequenziellen Layer-Gruppen **Technical/Abstract/Domain Interaction**, **Domain Service** und **Domain/Abstract/Technical Data**, andererseits gibt es die drei geschachtelten Layer-Gruppen **Technical Interaction/Data**, **Abstract Interaction/Data** und **Domain Interaction/Service/Data**.

Zusätzlich kann man in einer Anwendung 4 primäre Flüsse unterscheiden: der **Action Flow** läuft konsequent nur von oben nach unten, da alle Aktionen oben vom Benutzer (oder Nachbarsystemen) abgestoßen werden; der **Error Flow** läuft konsequent nur entsprechend in der Gegenrichtung, also von unten nach oben, da Fehler im Worst-Case dem Benutzer gemeldet werden müssen; der (fachliche) **Data Flow** und der (technische) **State Flow** laufen hingegen in beiden Richtungen, da Daten und Zustände sowohl persistiert als auch angezeigt werden müssen.

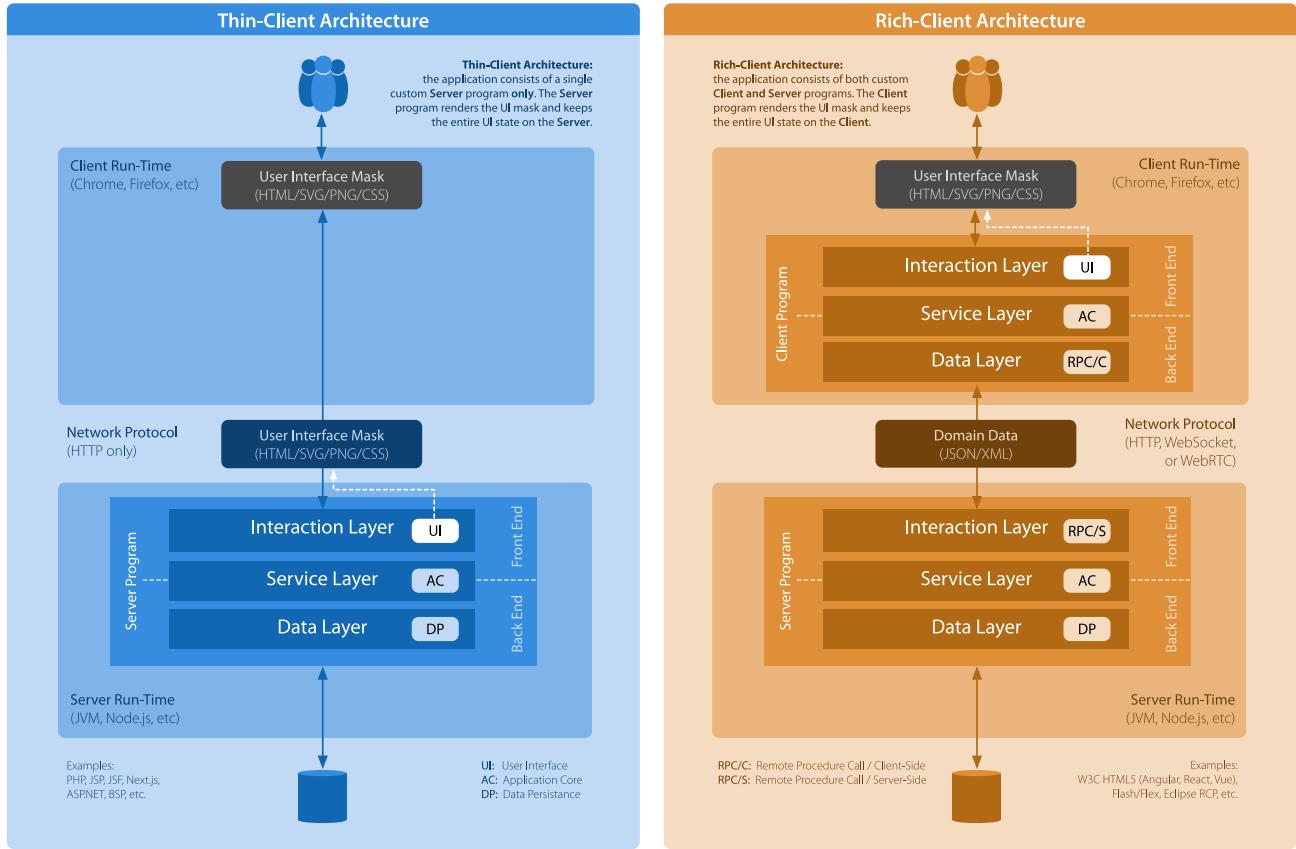
Die Abstraktion bei Interaction/Data in den Layern nimmt von oben/unten zur Mitte hin zu, weshalb auch dort der meiste fachliche Code einer Anwendung geschrieben wird. Für die oberen/unteren Layer wird üblicherweise massiv auf Open Source Libraries/Frameworks gesetzt.

Wenn man statt einem logischen Schnitt (der in einem **Internal Flow** zwischen den Layern resultiert) zwischen zwei Layern einen physikalischen Schnitt macht (der dann in einem **External Flow** resultiert), also die Anwendung in einzelne Programme auf unterschiedlichen Rechnern verteilt, so nennt man die daraus entstehende Architektur nach dem Umfang und der Verantwortung des Clients.

Bei **Thin Client** wird nur die **Technical Interaction** auf den Client ausgelagert, bei **Rich Client** wird die gesamte Benutzeroberfläche (also alle drei Layer **Technical/Abstract/Domain Interaction**) autonom auf den Client ausgelagert (üblicherweise als eine sog. "HTML5 Single-Page-Application"), bei **Fat Client** gibt es gar keinen zugehörigen Server mehr und bei der **Standalone** Application gibt nur noch ein einziges Programm.

## Fragen

- ? Wie nennt man die Anwendungs-Architektur, bei der die gesamte Benutzeroberfläche autonom auf dem Client läuft, während der Server nur rein fachliche Services liefert?
  - ? Wie heißen die Web-Anwendungen, welche eine **Rich Client** Architektur umsetzen?



Bei der **Thin-Client-Architektur** besteht die Anwendung aus nur einem einzigen individuellen Server-Programm. Dieses Server-Programm erzeugt die Maske der Benutzeroberfläche und hält den gesamten Zustand der Benutzeroberfläche auf dem Server.

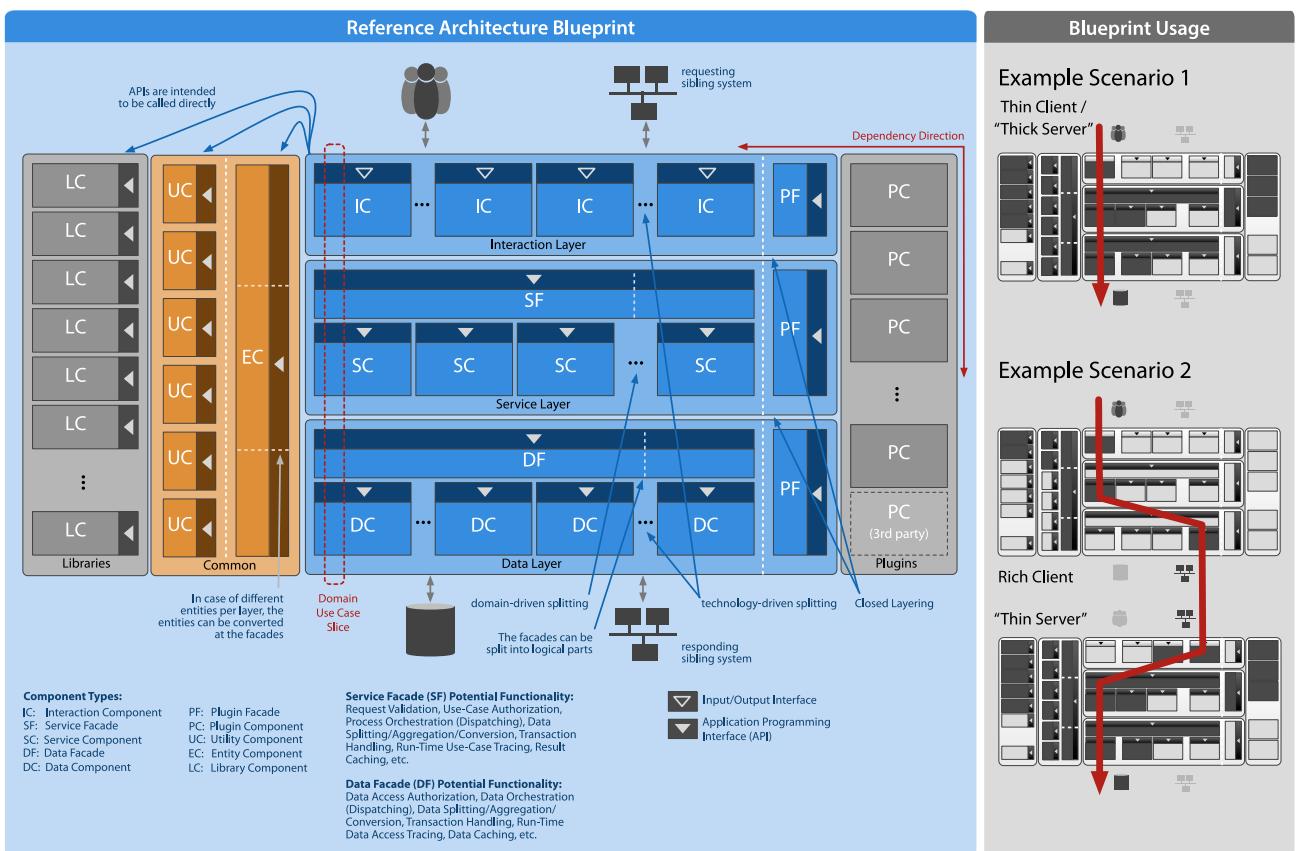
Der Vorteil dieser Architektur ist, daß die Anwendung sehr leicht aktualisiert werden kann. Der Nachteil dieser Architektur ist, daß die Benutzeroberfläche träge reagiert und der Zustand der Benutzeroberflächen aller Clients auf dem Server gehalten werden muss, was den Server zum Flaschenhals werden lassen kann.

Bei der **Rich-Client-Architektur** besteht die Anwendung sowohl aus einem individuellen Client- als auch einem Server-Programm. Das Client-Programm erzeugt die Maske der Benutzeroberfläche und hält den gesamten Zustand der Benutzeroberfläche auf dem Client.

Der Vorteil dieser Architektur ist, daß die Benutzeroberfläche eine hohe Reaktionsfähigkeit zeigt, nur fachliche Daten zwischen Client und Server ausgetauscht werden müssen und der Server weniger stark zum Flaschenhals wird. Der Nachteil dieser Architektur ist, daß gegebenenfalls der Client explizit über eine Installationsprozedur aktualisiert werden muss.

## Fragen

- Bei welcher Client-Architektur bietet die Benutzeroberfläche die höhere Reaktionsgeschwindigkeit?



Ein (betriebliches) Informationssystem folgt üblicherweise einer stringenten Komponenten-basierten Referenz-Architektur. Diese wird "full blown" dargestellt und kann beliebig "abgespeckt" werden.

Zuerst besteht diese Referenz-Architektur aus 3 wesentlichen Layern: dem **Interaction Layer** mit den (technisch geschnittenen) Komponenten, welche die I/O-basierten Schnittstellen zum Benutzer (User Interface) und/oder anfragenden Nachbarsystemen (über Network Interface) bereitstellen, dem **Service Layer** mit den (fachlich geschnittenen) Service-Komponenten (auch Anwendungskern genannt) und dem **Data Layer** mit den (technisch geschnittenen) Komponenten, welche die Anbindung an die eigene Datenbank und/oder abzufragenden Nachbarsystemen realisieren.

Man beachte, daß die "Andock-Position" eines Nachbarsystems von seinen Rollen abhängt: wenn es anfragt, dockt es am Interaction Layer an; wenn es abgefragt wird, dockt es am Data Layer an. Wenn es zufällig beide Rollen innehaben sollte, dockt es zweifach an. Die andere Sichtweise ist, daß sowohl der Benutzer als auch die Datenbank als spezielle "Nachbarsysteme" begriffen werden können.

Um die N **Interaction Components (IC)** mit M **Service Components (SC)** zu verbinden, wird üblicherweise eine entkoppelnde **Service Facade (SF)** eingezogen. Aus gleichem Grund gibt es üblicherweise auch eine **Data Facade**.

Das Datenmodell wird in gemeinsame **Entity Components (EC)** ausgelagert. Zusammen mit ggf. gemeinsam genutztem Code lebt beides in einem **Common Slice**. Libraries und Plugins sind ebenfalls in eigene Slices ausgelagert, es gibt aber zwei wesentliche Unterschiede: Libraries sind passiv und bieten der Anwendung ihre Funktionalität über ihre Schnittstellen an. Plugins sind aktiv und steuern die Anwendung, in dem sie sich über Service-Provider Interfaces (SPI) in den **Plugin Facades** in die Anwendung einklinken.

In der Anwendung darf es nur genau eine **Dependency Direction** geben, damit die Anwendung (in der Gegenrichtung der Dependencies) sauber gebaut werden kann. Die Referenz-Architektur wird außerdem üblicherweise zweifach instanziert, um sowohl einen Rich-Client als auch einen zugehörigen "Thin-Server" daraus zu konstruieren.

## Fragen

- ?
- Mit welchem Layer-Pattern werden in einem Informationssystem die **Interaction Components** von den **Service Components** entkoppelt?
- ?
- In welcher Reihenfolge werden die Komponenten einer Anwendung gebaut?

**Architecture & Systems**

**DEF** **Definition**

*Reactive System Architecture enables the realization of Reactive Systems.*

Reactive Systems are in **subordinated interaction** with their **dominating environment**. They **continuously process endless data streams as small messages**, react at **any time** and respond within **tight time limits**. For this, they continuously **observe their environment** and adapt their **behaviour** to the current situation.

**Demand & Deliverables**

**CTX** **Context**

*Real-time communication in the context of Digitization, Internet, Internet of Things (IoT), Systems of Engagement, Media and Analytics.*

**VAL** **Values**

*Non-blocking input/output data processing, fast responses within tight time limits, and continuous availability of the provided services.*

**REQ** **Requirements**

*Services are elastic and provide high scalability, and are **resilient** and provide high fault tolerance.*

**PRP** **Properties**

*Services run fully **autonomously**, monitor themselves, and automatically adapt to changes in the environment.*

**Principles**

**Stay Responsive**  
Always respond in a timely manner.

**Accept Uncertainty**  
Build reliability despite unreliable foundations.

**Embrace Failure**  
Expect things to go wrong and design for resilience.

**Assert Autonomy**  
Design components that act independently and interact collaboratively.

**Tailor Consistency**  
Individualize consistency per component to balance availability and performance.

**Decouple Time**  
Process asynchronously to avoid coordination and waiting.

**Decouple Space**  
Create flexibility by embracing the network.

**Handle Dynamics**  
Continuously adapt to varying demand and resources.

**Patterns & Paradigms**

**ARC** **Architecture**

Microservices, Cloud-Native Architecture (CNA), Event-Driven Architecture (EDA).

**COM** **Communication**

Asynchronous Communication, Non-Blocking I/O, Sequence, Push, Backpressure, Quality of Service (QoS).

**DAT** **Data**

Semantical Event, Small Message, Endless Stream.

**STY** **Style**

Functional Programming, Asynchronous Programming.

**EXE** **Execution**

Parallelization, Concurrency, Actors, Threads, Thread-Pool, Event-Loop.

**INF** **Infrastructure**

Message Queue (MQ), Load Balancer, Reverse Proxy, Service Mesh, Virtual Private Network (VPN).

**PRC** **Processing**

Complex Event Processing (CEP), EAI Patterns, Stream Processing (map, flatMap, filter, reduce), Event Sourcing.

**ASY** **Asynchronism**

Callback, Promise/Future, Observable, Publish & Subscribe.

**Reactive System Architecture** erlaubt die Realisierung von **Reactive Systems**. Reactive Systems sind in **untergeordneter Interaktion** mit ihrer **dominierenden Umgebung**. Sie verarbeiten **kontinuierlich endlose Datenströme** in Form von **kleinen Nachrichten**, reagieren zu **jeder Zeit** und antworten innerhalb **enger Zeitgrenzen**. Dazu **beobachten** sie kontinuierlich ihre **Umgebung** und **passen ihr Verhalten** an die aktuelle Situation an.

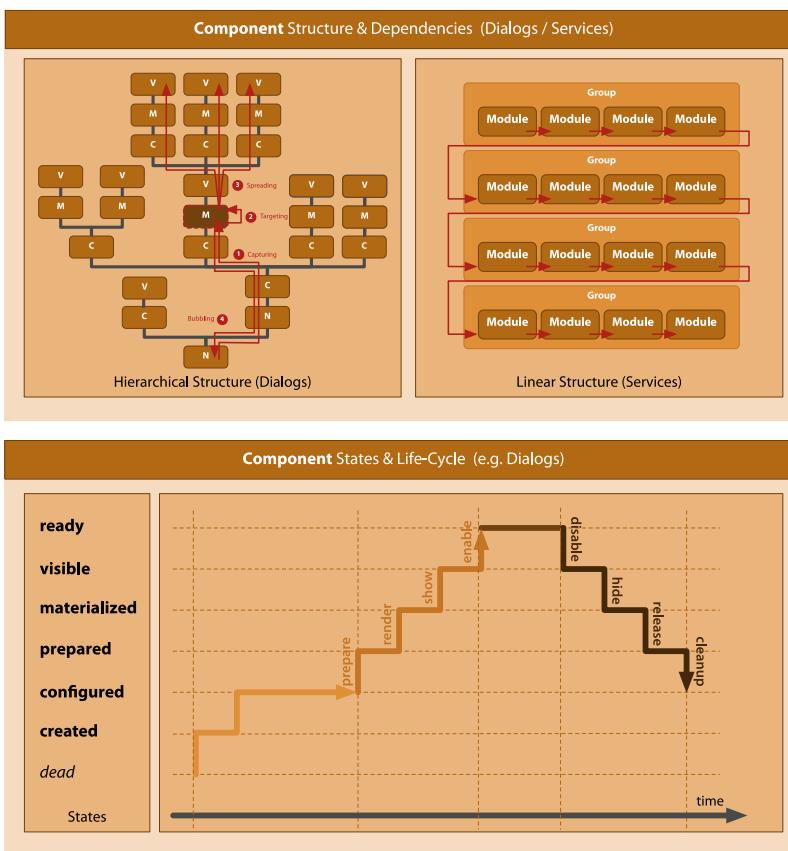
Reactive Systems werden primär im Kontext von **Echtzeit-Kommunikation** verwendet, wo Dienste bereitgestellt werden, die **elastisch** (**elastic**) sein und hohe Skalierbarkeit anbieten müssen und die **verlässlich** (**reliable**) sein und hohe Fehlertoleranz anbieten müssen.

## Fragen

- ?
- Welche zwei wesentlichen Anforderungen erfüllen Reactive Systems?
  
- ?
- Was charakterisiert Reactive Systems in Bezug auf ihre Datenverarbeitung?

**AF**  
10.4

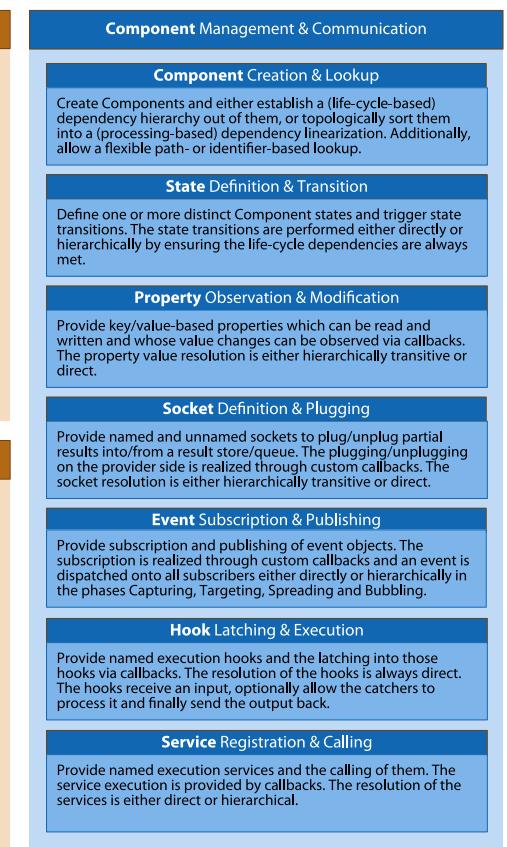
Printed or otherwise stored, when 1.11.2020, Author: Dr.-Ing. Dipl.-Ing. Dr.-Ing. h. c. mult. Michael Schäfer, Chair of Micro- and Nanoelectronics, Institute of Technology Research and Innovation, University of Münster. Unauthorized reproduction is prohibited.



Um das Maxim "Component Orientation" in der Praxis sinnvoll umsetzen zu können, wird meist ein sogenanntes **Component System** eingesetzt. Dessen Funktionalität erlaubt die Verwaltung und die Kommunikation von einer Vielzahl von Komponenten und damit das Meistern der Gesamtkomplexität einer Anwendung.

Die Funktionalität eines **Component System** umfasst meist die folgenden wesentlichen Aspekte:

- Component Creation & Lookup** zur Strukturierung der Komponenten,
- State Definition & Transition** für den Lebenszyklus der Komponenten,
- Property Observation & Modification** für die Daten der Komponenten,
- [Result] Socket Definition & Plugging** für die Ergebnisse der Komponenten,
- Event Subscription & Publishing** für Ereignisse von Komponenten,
- Hook Latching & Execution** für die Verbindung zwischen Komponenten und
- Service Registration & Calling** für die Aufrufe zwischen Komponenten.

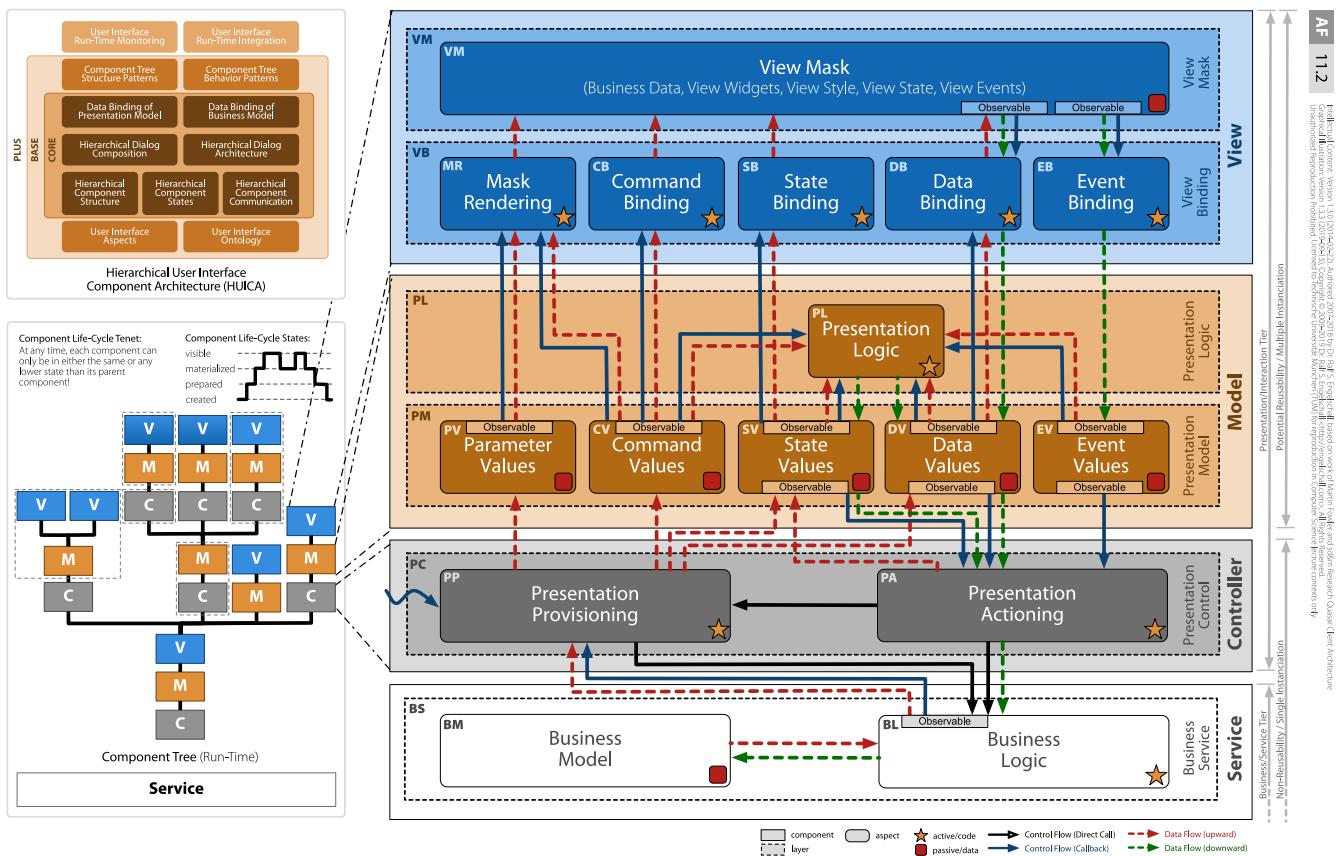


Es gibt primär zwei Arten von Komponenten-Strukturen und -Abhängigkeiten, die ein **Component System** unterstützt: eine **Hierarchical Structure** für Clients (da User Interfaces inhärent hierarchisch sind) und **Linear Structure** für Server (da die Request/Response-Verarbeitung inhärent linear ist).

Für den Lebenszyklus von Komponenten erlaubt ein **Component System** meist die Definition von beliebigen Zuständen, welche über die Zeit von Komponenten eingenommen werden können und welche vom **Component System** zentral verwaltet werden.

## Fragen

- ❓ Wie kann man die Gesamtkomplexität für die Verwaltung und die Kommunikation von einer Vielzahl von Komponenten beim Maxim **Component Orientation** meistern?
- ❓ Welche zwei Arten von Komponenten-Strukturen werden meist von **Component Systems** unterstützt?

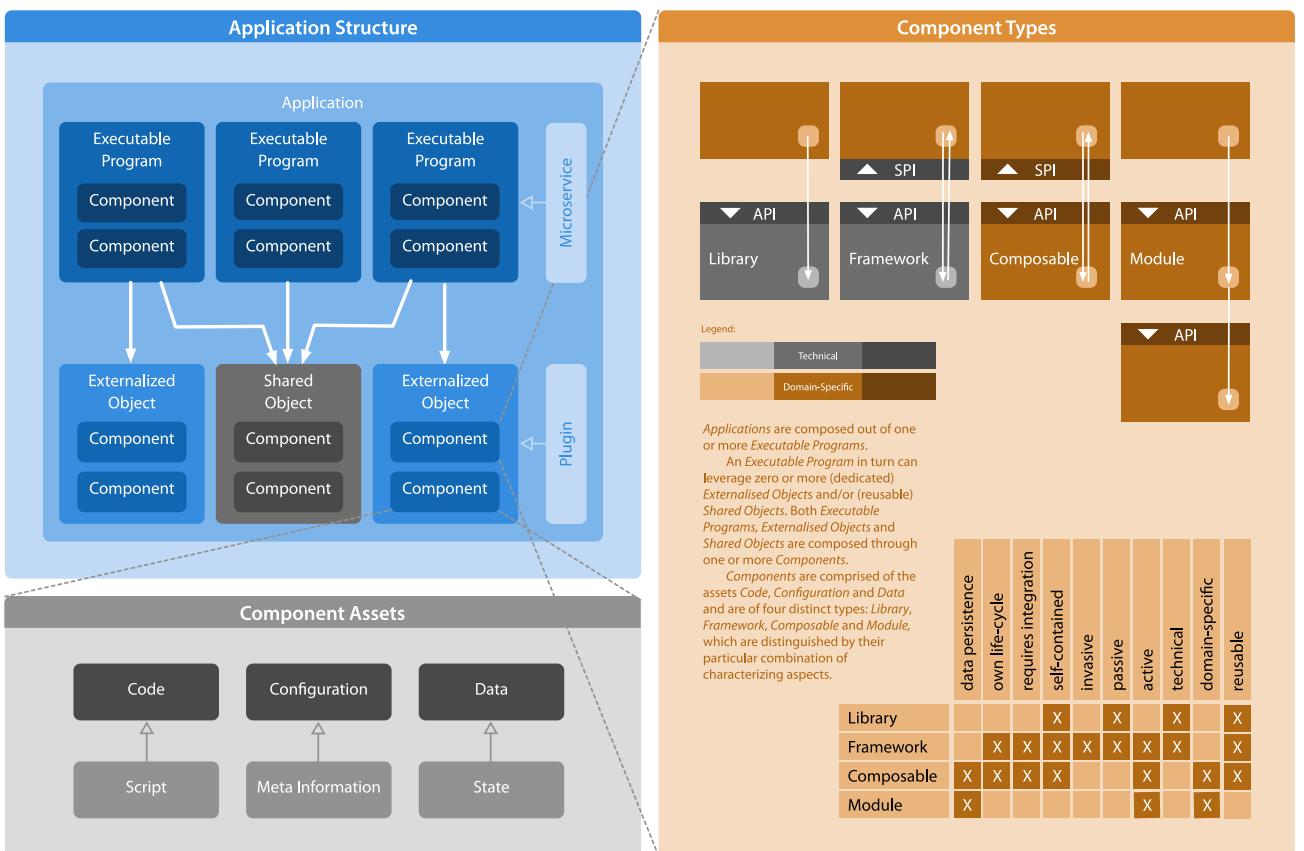


FIXME

## Fragen

?

FIXME



Anwendungen werden aus einem oder mehreren Executable Programs zusammengesetzt. Ein Executable Program kann seinerseits null oder mehrere (dedizierte) Externalized Objects und/oder (wiederverwendbare) Shared Objects nutzen. Sowohl Executable Programs, Externalized Objects und Shared Objects bestehen aus einer oder mehreren Components. In einer Microservice Architecture werden die Executable Programs als Microservices bezeichnet. In einer Plugin Architecture werden die Externalized Objects als Plugins bezeichnet.

Es gibt vier verschiedene Typen von Components: Library (Bibliothek), Framework, Composable und Module (Modul). Sie lassen sich durch ihre besondere Kombination von charakteristischen Aspekten unterscheiden. Am wichtigsten ist, ob sie ein Application Programming Interface (API) für den Konsumenten der Component bereitstellen und/oder ob sie verlangen, dass der Konsument der Component eine Art von Service Provider Interface (SPI) erfüllen muß.

## Fragen

- ❓ Was ist der wesentliche Unterschied zwischen einer Library und einem Framework?