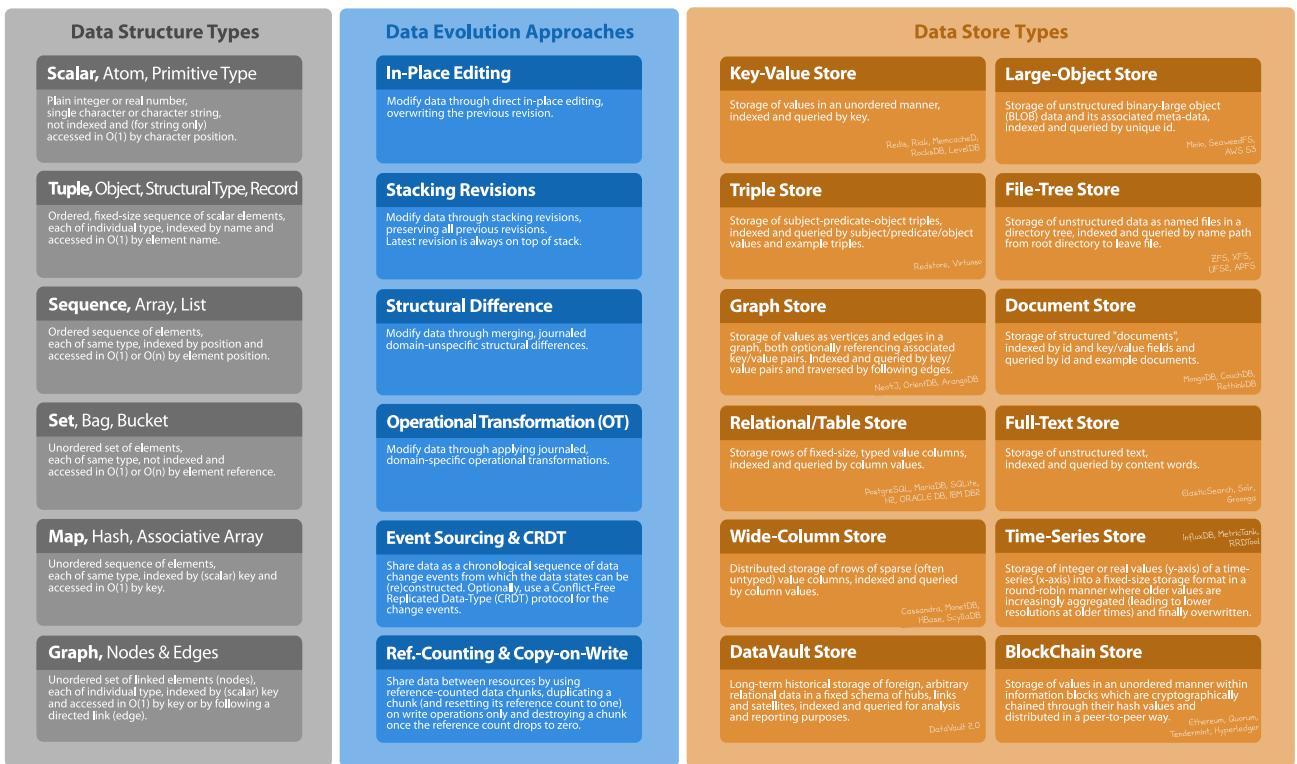




Software Engineering in der industriellen Praxis (SEIP)

Dr. Ralf S. Engelschall



Der Software Architect unterscheidet nur 6 **Data Structure Types** für Daten-Elemente: **Scalar** (z.B. Integer, String, etc), **Tuple** (ordered fixed-size sequence of Scalars), **Sequence** (ordered sequence of elements), **Set** (unordered set of elements), **Map** (unordered set of elements, each indexed by key) und **Graph** (unordered set of elements, each indexed by key or by following a link between elements). Alle komplexen spezifischen Datenstrukturen in der Praxis sind für den Software Architect nur die Kombination dieser 6 Typen.

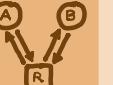
Es gibt zahlreiche **Data Evolution Approaches**, mit denen sich Daten über die Zeit verändern können: im einfachsten Fall, **In-Place Editing**, werden Daten einfach direkt geändert. Ein Zugriff auf frühere Stände gibt es hierbei nicht. Soll auf frühere Stände zugegriffen werden können, so kann man **Stacking Revisions** einsetzen, bei dem vor jeder Änderung der gesamte Datensatz zuerst kopiert wird. Damit nicht der gesamte Datensatz kopiert werden muss, wird bei **Structural Difference** nur eine technische Differenz von altem und neuem Datensatz gespeichert. Alternativ können bei **Operational Transformation** die fachlichen Änderungsoperationen als Journal gespeichert werden.

Wird so ein Journal benutzt, um auch Replika der Datensätze aktuell zu halten, spricht man von **Event Sourcing**. Falls das Journal als dem Protokoll von sogenannten **Conflict-Free Replicated Data-Types (CRDT)** aufgebaut ist, lässt sich statt (unidirektionaler) Replikation auch eine (bidirektionale) Synchronisation erzielen. Falls mehrere konkurrierende Prozesse/Threads logisch auf Kopien, aber physikalisch auf denselben Datensätzen operieren, lässt sich mit **Copy-on-Write** und **Reference Counting** ein gemeinsamer Zugriff und der Lebenszyklus der Datensätze dennoch sinnvoll steuern.

Zur Speicherung von Daten in Datenbanken gibt es zahlreiche **Data Store Types**. Diese unterscheiden sich primär in der Art und Flexibilität der Datenstruktur und den gewährten Garantien. Der verbreitetste Typ ist der **Relational/Table Store**. Der eleganteste Typ ist der **Graph Store**. Der bequemste ist der **Document Store**.

Fragen

- ?
- Nennen sie 3 **Data Evolution Approaches**, die es jeweils erlauben, auf die früheren Stände der Daten zuzugreifen?

Data Guarantees	Data Access	Data Spreading & Aggregation
CAP (Trade-In) A distributed data store cannot provide more than two out of three guarantees: Consistency (C), Availability (A), Partition-Tolerance (P). So, it has to choose between Consistency (CP) and Availability (AP) when a network partition or failure happens. 	Shared Read/Write Shared access to data for both read and write operations. Example: Multiple threads in a NoSQL database setup. 	Data River (1-to-N) A real-time fan-out replication of data from a single upstream/source data repository to multiple downstream/target data repositories. 
BASE (NoSQL) The semantics: B ounded by Network (B), A vailability (A), E ventual consistency. BASE systems favor Availability over Consistency in the CAP-context. 	Shared Read / Exclusive Write Shared access to data for read operations and exclusive access to a single "writing" component to data for write operations. Example: RDBMS Master-Slave cluster with shared storage. 	Data Mart (N-to-1), ODS A massive, shared, easily accessible data repository for storing "big data" from many upstream sources in a (real-time) and structured way and with knowing the actual subsequent analysis usage. 
ACID (RDBMS, NewSQL) The four guarantees provided in parallel (usually by RDBMS and NewSQL systems): Atomicity, Consistency, Isolation and Durability. ACID systems usually favor Consistency over Availability in the CAP-context. 	Shared Nothing No shared access to data at all for both read and write operations. Example: Leader-Follower setup with RAFT consensus where Leader writes data only. 	Data Lake (N-to-1), Cache A massive sized, easily accessible data repository for storing "big data" from many upstream sources in a (real-time) semi-structured way and without knowing the actual subsequent usage. 
Data Access Grouping	Data Consistency	Data Transfer
Transaction Protect a sequence of operations from interim exceptions by bracketing the operations in a technical transaction (ensuring that either all or none of the operations succeed). 	Exclusive Locking (Mutex) Protect data from concurrent access and resulting inconsistencies with a mutual exclusion lock (mutex) which allows just a single peer to access the data at a time. 	Replication Continuously stream or regularly copy data from a master system to one or more slave systems in order to either have data on slave systems or have slave systems available as a fallback/backup in case of a failure of the master system. 
Compensation Protect a sequence of operations from interim exceptions by undoing the already succeeded operations through domain-specific compensating (reverse) operations. 	Optimistic Locking Protect data from concurrent access and resulting inconsistencies by taking note of a revision number or content hash during read operations and checking that this information has not changed before writing the data. 	Synchronization Continuously stream or regularly copy data between multiple master systems and resolve potential concurrent data modification conflicts. This may allow distributed and even disconnected computing. 

Im Bereich der **Data Guarantees** gibt es drei wesentliche Aspekte: Das **CAP Theorem** adressiert den sog. "Trade-In": Man muss sich in der Praxis üblicherweise zwischen Consistency + Partition-Tolerance (CP) oder Availability + Partition-Tolerance (AP) entscheiden. Beides gleichzeitig geht nicht. Bei **BASE**-Systemen favorisiert man üblicherweise AP. Bei einem traditionellen RDBMS mit **ACID** Garantien favorisiert man üblicherweise CP.

Beim **Data Access Grouping** kennt man **Transaction** und **Compensation**. Ersteres ist eine "technische Klammer", die einem erlaubt, in Fehlerfall wieder zu dem früheren Zustand zurückzukehren. Letzteres ist eine "fachliche Klammer", bei der sog. Kompensationsoperationen einem erlauben die früheren Änderungen zu "stornieren", um so einen früheren konsistenten Zustand wiederzuerlangen.

Beim **Data Access** von zwei oder mehr Prozessen/Threads auf die gleichen Daten unterscheidet man die Ansätze **Shared Read/Write** (alle lesen und schreiben dieselben Daten), **Shared Read / Exclusive Write** (alle lesen und nur einer schreibt dieselben Daten) und **Shared Nothing** (alle lesen und schreiben auf die gleichen synchronisierten Daten).

Bei der **Data Consistency** kennt man **Exclusive Locking** (pro Zeiteinheit schreibt nur einer) und **Optimistic Locking** (alle versuchen zu schreiben, erkennen und lösen aber einen Konflikt).

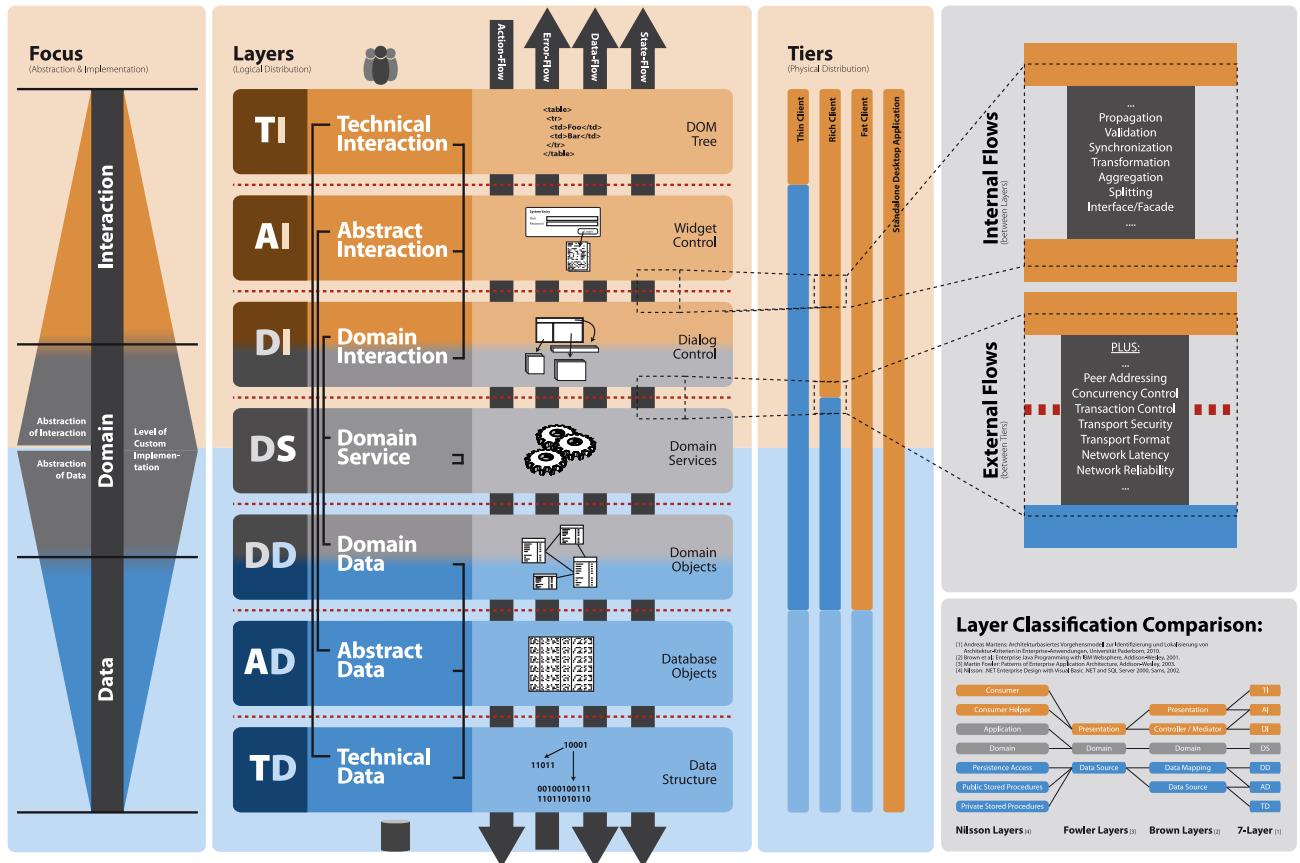
Beim **Data Spreading & Aggregation** unterscheidet man drei Arten: beim **Data River** werden die Daten von einem Master-System auf viele Slave-Systeme repliziert, um u.a. eine höhere Read-Performance zu erzielen. Beim **Data Mart** (strukturierte Daten) und **Data Lake** (semi-strukturierte Daten) werden dagegen die Daten von vielen Master-Systemen auf ein Slave-System repliziert, um u.a. die Daten zentral auswerten oder "cachen" zu können.

Beim **Data Transfer** wird zuletzt noch zwischen der unidirektionalen und konfliktfreien **Replication** und der bidirektionalen und konfliktreichen **Synchronisation** unterschieden.

Frage

- ?

 - Wie nennt man den Ansatz, bei dem Daten von einem Master-System auf viele Slave-Systeme repliziert werden?



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.

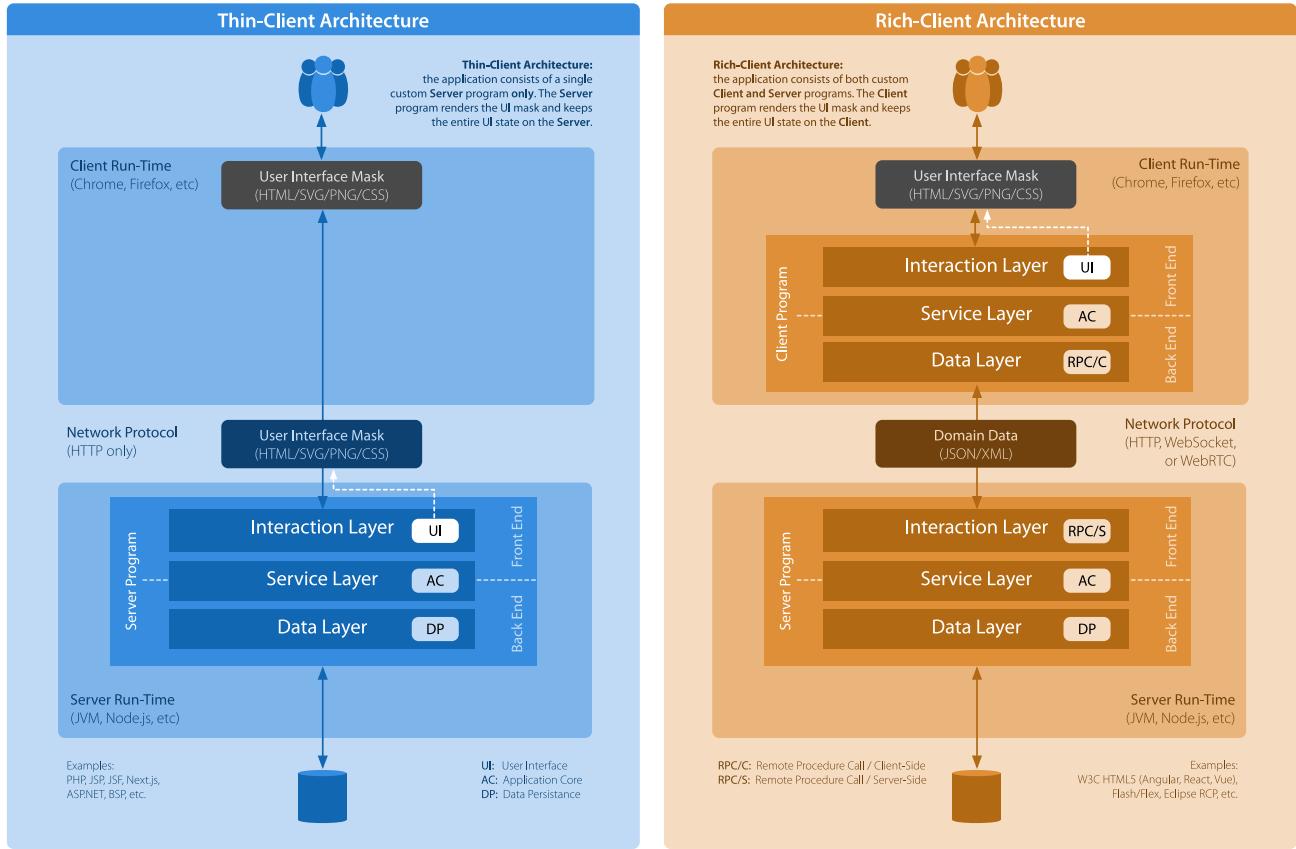
Die Abstraktion bei Interaction/Data in den Layern nimmt von oben/unten zur Mitte hin zu, weshalb auch dort der meiste fachliche Code einer Anwendung geschrieben wird. Für die oberen/unteren Layer wird üblicherweise massiv auf Open Source Libraries/Frameworks gesetzt.

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.

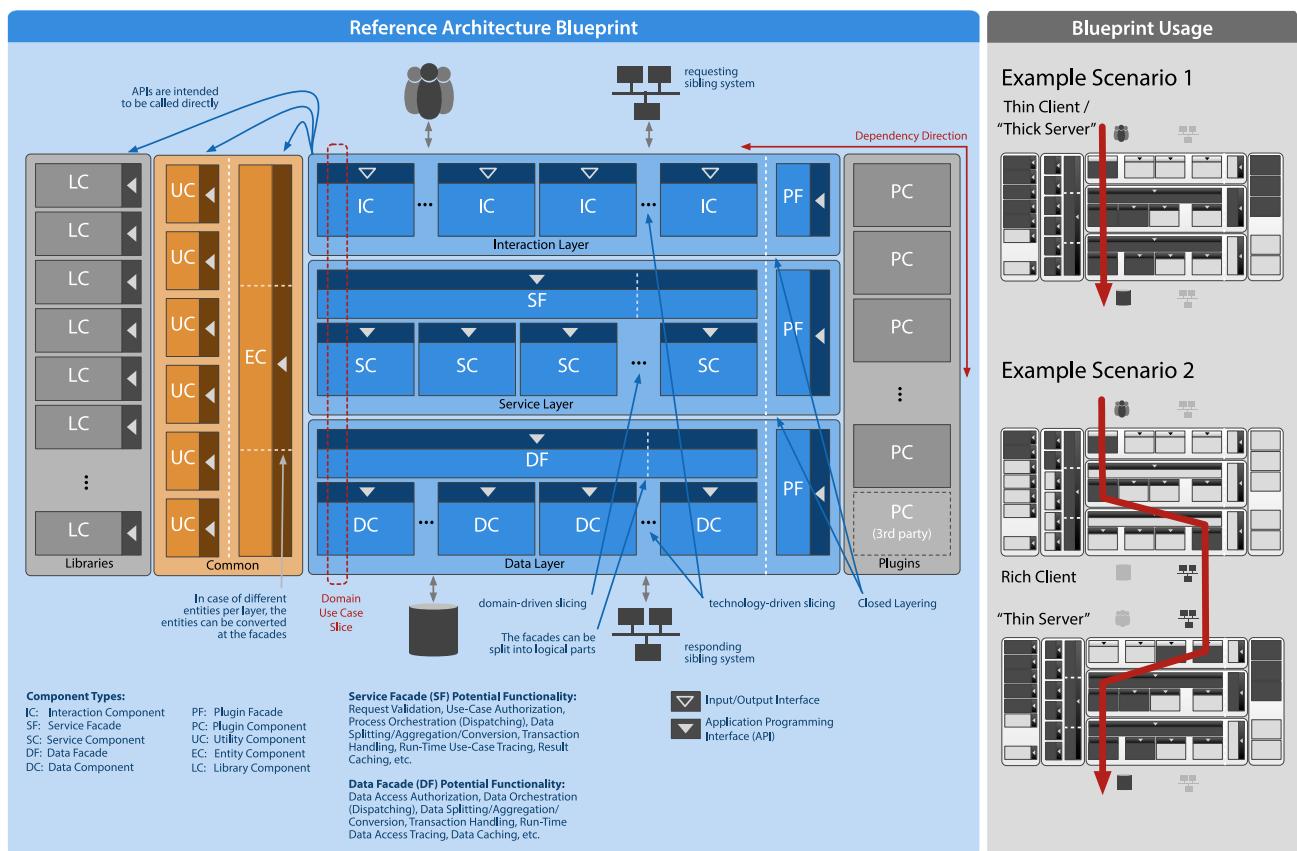
Bei der **Rich-Client-Architektur** besteht die Anwendung sowohl aus einem individuellen Client- als auch einem Server-Programm. Das Client-Programm erzeugt die Maske der Benutzeroberfläche und hält den gesamten Zustand der Benutzeroberfläche auf dem Client.

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?