SKS-5, WS 2018
Bernhard Löwenstein

Softwarekomponentensysteme

Präsentation

Agenda

- Software Architecture
- Key Principles
- Architectural Styles
- Application Archetypes
- Architectural Documentation
- Design Patterns (dt. Entwurfsmuster)
- Software Components
- Web Services
- SOAP Web Services
- RESTful Web Services
- Microservices

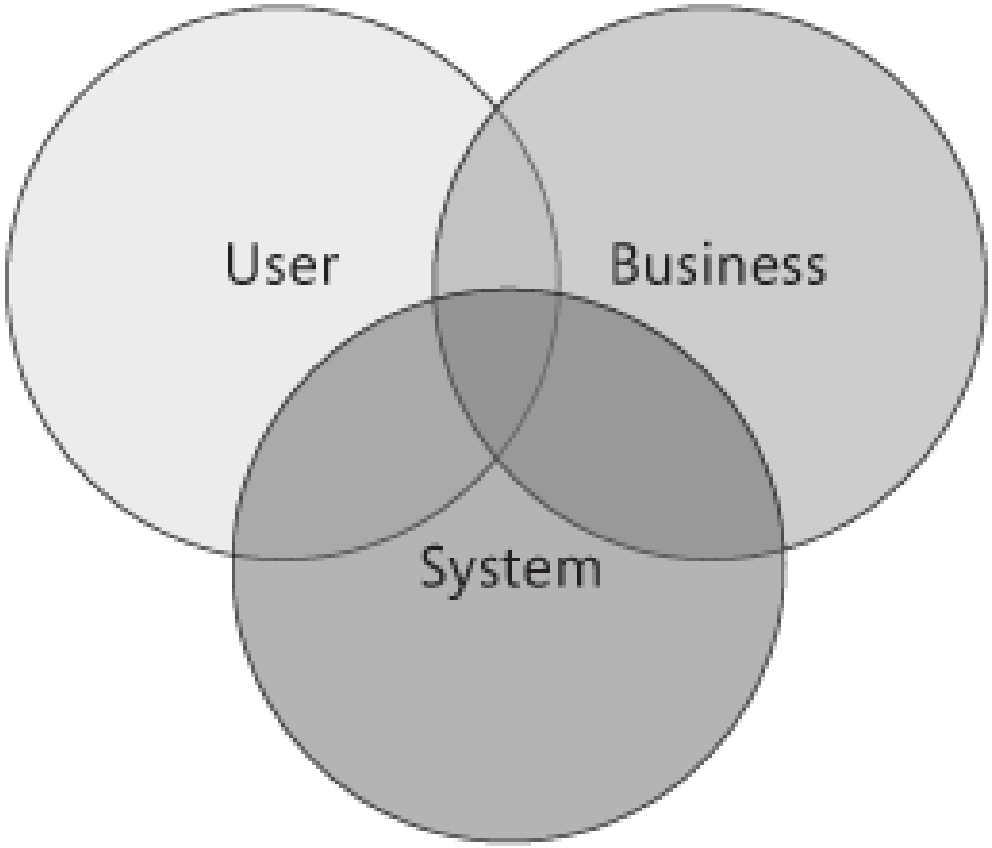
Software Architecture

What is Software Architecture?

"Software architecture encompasses the set of significant decisions about the organization of a software system including the selection of the structural elements and their interfaces by which the system is composed; behavior as specified in collaboration among those elements; composition of these structural and behavioral elements into larger subsystems; and an architectural style that guides this organization. Software architecture also involves functionality, usability, resilience, performance, reuse, comprehensibility, economic and technology constraints, tradeoffs and aesthetic concerns."

(Mary Shaw & David Garlan)

Why is Architecture Important?



Key Principles

Key Architecture Principles

- Build to change instead of building to last
- Model to analyze and reduce risk
- Use models and visualizations as a communication and collaboration tool
- Identify key engineering decisions

Key Design Principles

- Separation of concerns
- Single responsibility principle
- Don’t repeat yourself (DRY)
- Minimize upfront design

Separation of Concerns

- *"Separating a computer program into distinct features that overlap in functionality as little as possible"*
- Examples:
  - OSI Layer Stack
  - HTML, CSS, JavaScript
  - MVC
  - Aspected-Oriented Programming

Single Responsibility Principle

- *"Every class should have a single responsibility, and that responsibility should be entirely encapsulated by the class"*
- Example:
  - Format & Print Document
  - 2 Classes: Formatter, Printer

Don’t Repeat Yourself (DRY)

- *"Every piece of knowledge must have a single, unambiguous, authoritative representation within a system"*
- Tools:
  - Methods, Classes, Libraries, …
  - Code Generators
  - Build Systems

Minimize Upfront Design

- *"Only design what is necessary"*
- You Ain't Gonna Need It (YAGNI)
- The time spent is taken from adding, testing or improving necessary functionality.
- The new features must be debugged, documented, and supported.
- Any new feature imposes constraints on what can be done in the future, so an unnecessary feature now opens the possibility of conflicting with a necessary feature later.
- Until the feature is actually needed, it is difficult to fully define what it should do and to test it.

Key Design Considerations

- Determine the Application Type
- Determine the Deployment Strategy
- Determine the Appropriate Technologies
- Determine the Quality Attributes
- Determine the Crosscutting Concerns

SOLID OOP Principle

S **Single responsibility principle**
an object should have only a single responsibility

O **Open/closed principle**
software entities should be open for extension, but closed for modification

L **Liskov substitution principle**
objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program

I **Interface segregation principle**
many client specific interfaces are better than one general purpose interface

D **Dependency inversion principle**
do not depend upon concretions, one should depend upon abstractions

Architectural Styles

Architectural Style

A set of principles that provide an abstract framework for a family of systems

Architectural Styles

Category	Architecture Styles
Communication	Service-Oriented Architecture (SOA) Message Bus
Deployment	Client/Server N-Tier (common: 3-Tier)
Structure	Component-Based Object-Oriented Layered Architecture

Service-Oriented Architectural Style

- Functionality as a set of services
- Standards based interfaces
- Services:
  - are autonomous
  - are distributable
  - are loosely coupled
  - share schema and contract, not classes
  - Compatibility is based on policy

Service-Oriented Architectural Style

- Principles:
  - Abstraction
  - Discoverability
  - Interoperability

Message Bus Architectural Style

- Ability to receive & send messages over channels
- Applications don't need to know each other
- Pluggable architecture: Systems attach and detach
- Asynchronously by design
- Common Bus = Router, Producer/Consumer, Publish/Subscribe patterns

Message Bus Architectural Style

- Variations:
  - Intra-System Message Bus (e. g. CAN)
  - Enterprise Service Bus (ESB)
  - Internet Service Bus (ISB)
- Pros:
  - Extensibility
  - Low complexity
  - Flexibility
  - Loose coupling
  - Scalability
  - Application simplicity

Architectural Styles

Category	Architecture Styles
Communication	Service-Oriented Architecture (SOA) Message Bus
Deployment	Client/Server N-Tier (common: 3-Tier)
Structure	Component-Based Object-Oriented Layered Architecture

Client/Server Architectural Style

- Classically referred to as "2-Tier Architecture"
- UI Client to access a database with logic (triggers, stored procedures)
- Examples:
  - Websites: Browser ⇔ Web Server
  - FTP Clients

Client/Server Architectural Style

- Pros:
  - High security: central data store
  - Centralized data access
  - Ease of maintenance
  - …
- Cons:
  - Data & Business logic closely coupled
  - Does not scale
  - Hard to extend
  - …

N-Tier / 3-Tier Architectural Style

- Independent tiers
  - Only the direct neighbours know each other
- Partitions the concerns of the application into stacked groups (layers)
- Tiers communicate with each other via defined interfaces / communication channels
- Can be deployed on different machines

N-Tier / 3-Tier Architectural Style

- Pros:
  - Maintainability: Each tier is independent
  - Scalability: Tiers on different machines
- Cons:
  - More complex development efforts
  - Clear separation between tiers can be hard

Architectural Styles

Category	Architecture Styles
Communication	Service-Oriented Architecture (SOA) Message Bus
Deployment	Client/Server N-Tier (common: 3-Tier)
Structure	Component-Based Object-Oriented Layered Architecture

Component-Based Architectural Style

- Decomposition of design into functional or logical components
- Components expose well-defined interfaces with methods, events and properties
- Examples: UI components, business components, data access components
- Execution environment: COM, DCOM, CORBA, Java EE platform, …

Object-Oriented Architectural Style

- System consists of cooperating objects instead of routines or procedural instructions
- Principles:
  - Abstraction
  - Composition
  - Inheritance
  - Encapsulation
  - Polymorphism

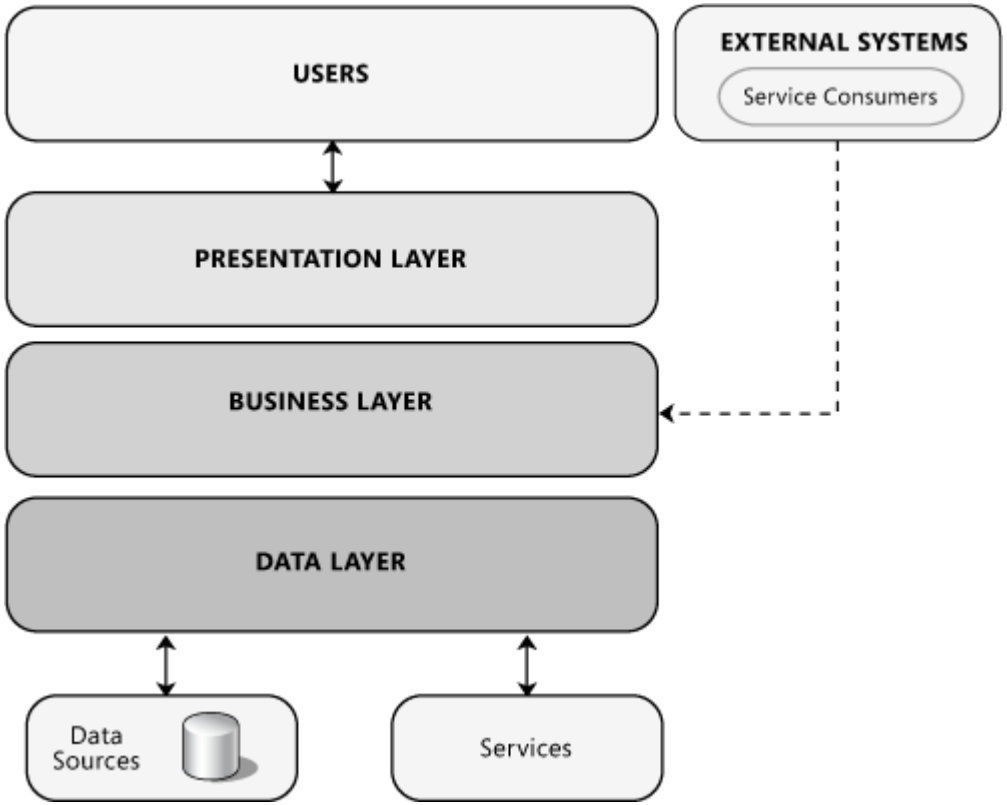
Layered Architectural Style

- Grouping of related functionality into distinct layers, stacked vertically on top of each other
- Functionality in each layer related by a common role or responsibility (e. g. presentation, business logic, data access)
- Communication between layers is explicit & loosely coupled
- Only dependencies on the N-1th layer allowed

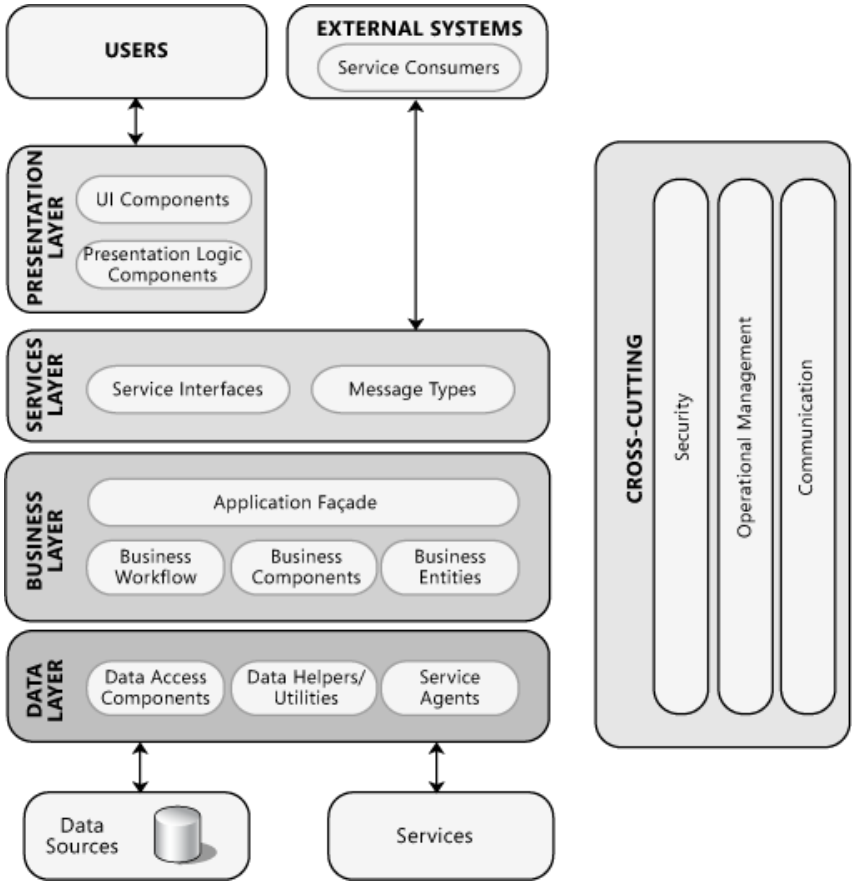
Layered Architectural Style

- Common Principles:
  - Abstraction
  - Encapsulation
  - Clearly defined functional layers
  - Reusable
  - Loose coupling

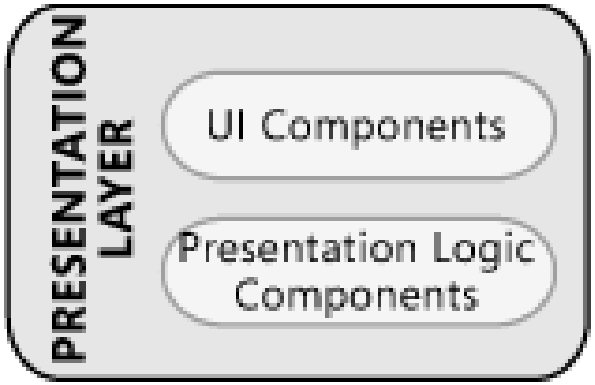
Logical View of 3-Layer Application



More Detailed Architecture



Presentation Layer



- UI Components
  - Visual Elements aka "Controls"
- Presentation Logic Components
  - Logical behavior of the UI
  - UI flow, Validation, Calculation, ..
  - Use appropriate patterns (Model-View-Controller, Model-View-Presenter, …)

Presentation Layer Technologies

- Swing
- JavaServer Faces
- JavaFX
- Windows Presentation Foundation / Silverlight
- Windows Forms
- Flash
- …

Challenges at Presentation Layer

- Validation
- Navigation
- Exception management
- Composition
- Communication
- Caching

Service Layer

- Important separation:
  - Hides internal implementation
- Service contracts
- Message types
- Translator components

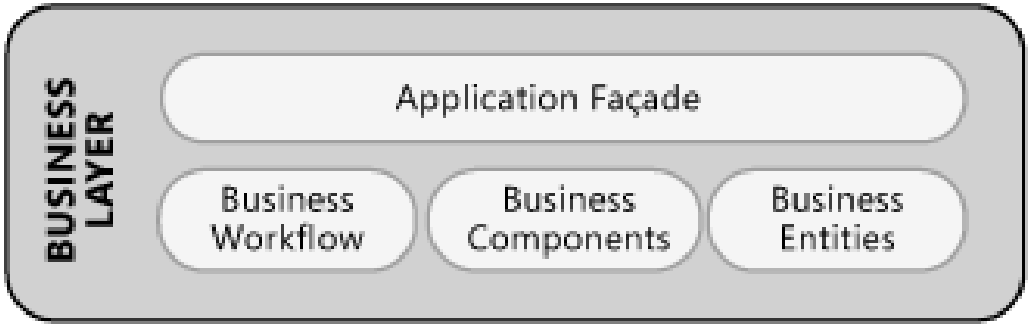
Services Design

- Application-scoped
- Extensibility
- Design service contracts separately
- Compose message types of standard elements
- Assume invalid requests
- Detect & manage repeated messages
- Manage messages arriving out of order

Service Layer Technologies

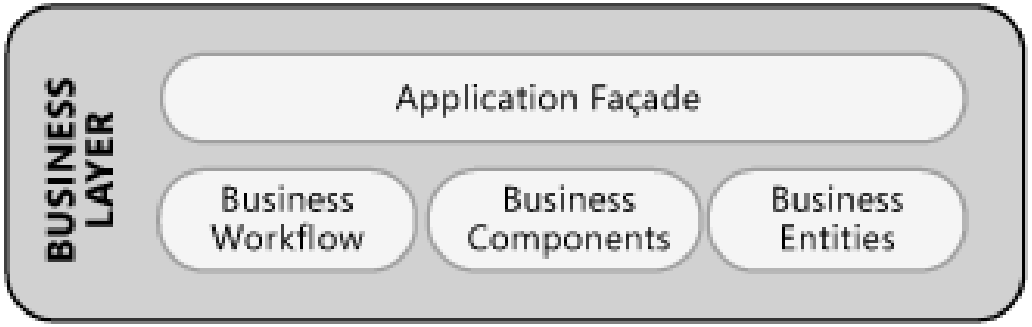
- Protocols:
  - SOAP, REST, …
- Frameworks / Infrastructure:
  - JAX-WS
  - JAX-RS
  - Apache Axis
  - Windows Communication Foundation
  - …

Business Layer



- Application façade:
  - Simplified interface
  - Hides complexity
  - Combines multiple operations & components
  - Just forwards calls, does security checks, …

Business Layer

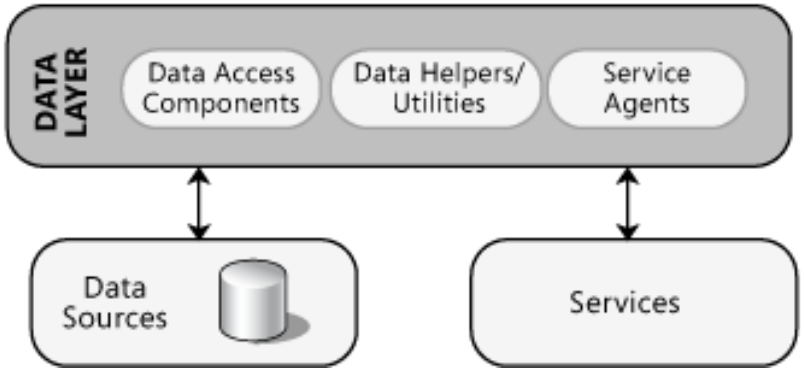


- Business logic components:
  - Business workflow components
  - Business components
  - Business entity components

Business Layer

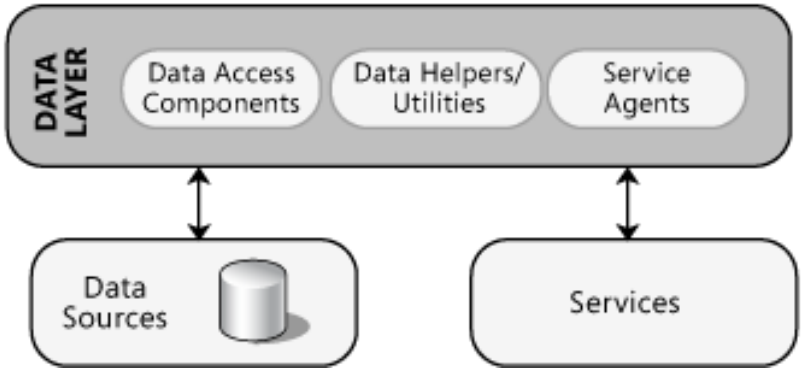
- Business workflow components:
  - Perform business processes
  - Orchestration between multiple steps
  - Long running processes
- Business components
  - Encapsulate logic, calculation, validation, …
- Business entity components
  - Store data (e. g. Customer, Order)
  - Sometimes also: validation & logic

Data Layer



- Access data sources of different kinds:
  - Relational
  - XML
  - DataSets
  - Services

Data Layer



- Data access components
  - Encapsulate data source specific access code
  - Grouped by source types
  - e. g. CustomerDbAccess, BookingXmlWriter
- Service agents
  - Encapsulate access to service proxies
  - Similar to data access components

Data Exchange via Layer Boundaries

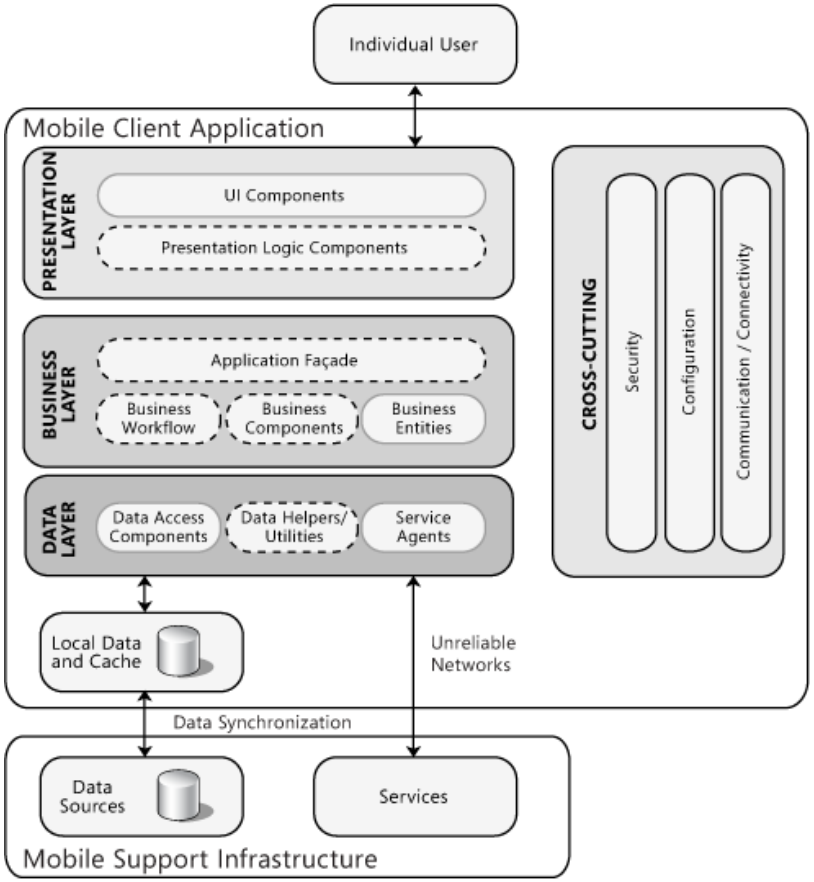
- Predefined contracts (interfaces)
- Known to both sides
- Contract versioning:
  - Side by side:
    - Multiple parallel deployments
  - Inheritance:
    - Single deployment
    - Old contracts still exist
  - Translators / Adapters:
    - Translate old format to new one
- Technology independent DTOs

Application Archetypes

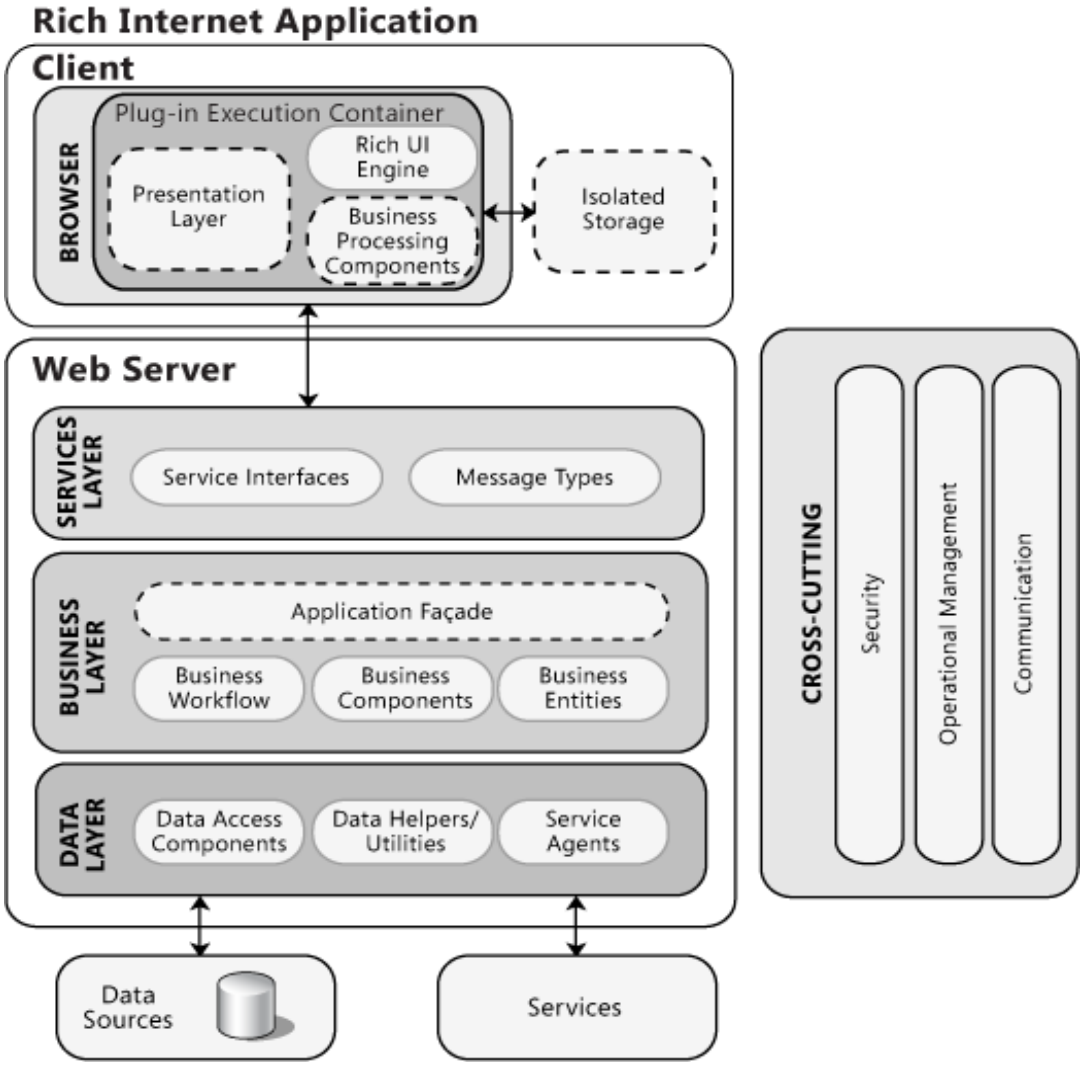
Common Application Archetypes

- Mobile applications
- Rich Internet applications
- Service applications
- Web applications

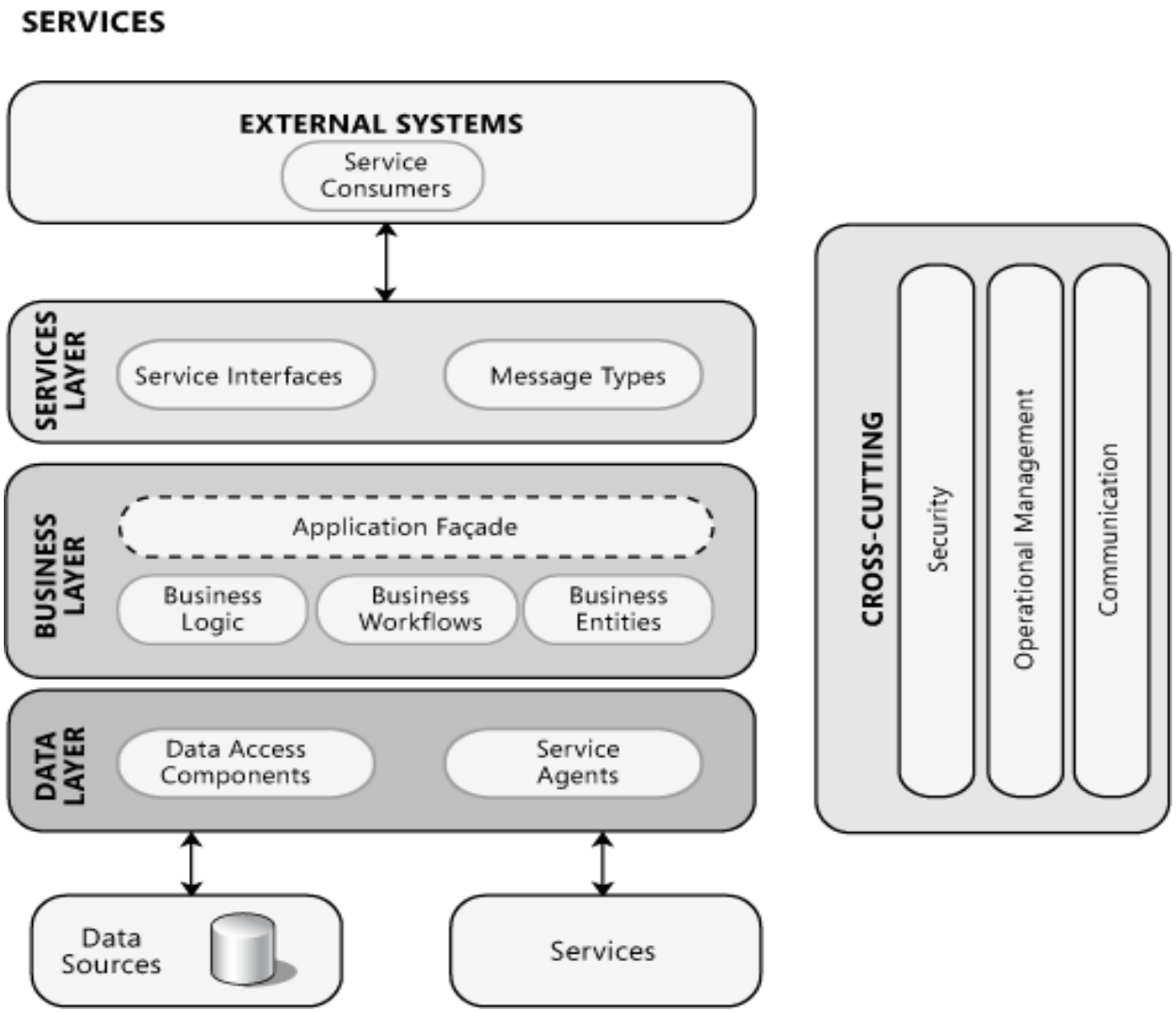
Mobile Application



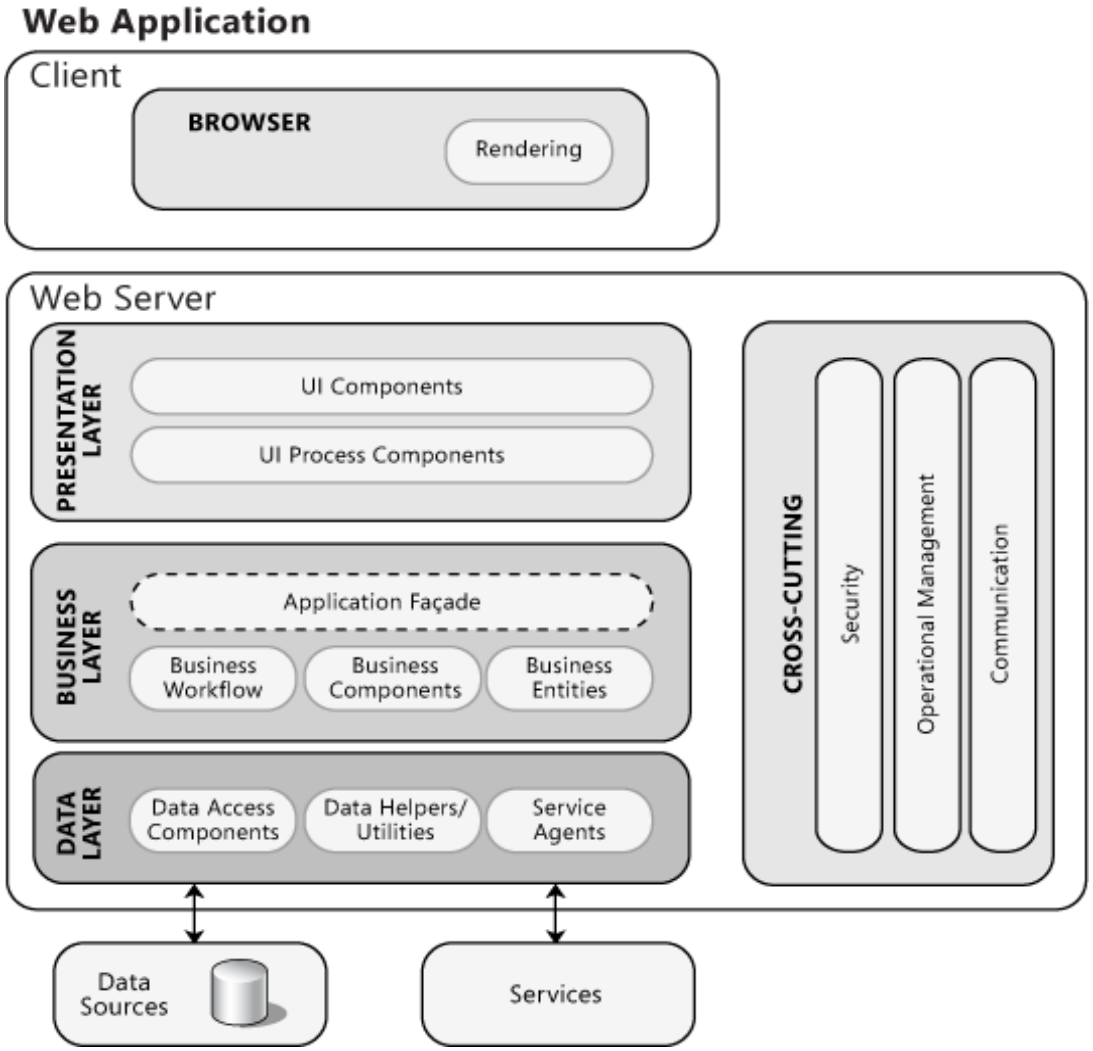
Rich Internet Application



Service Application



Web Application



Architectural Documentation

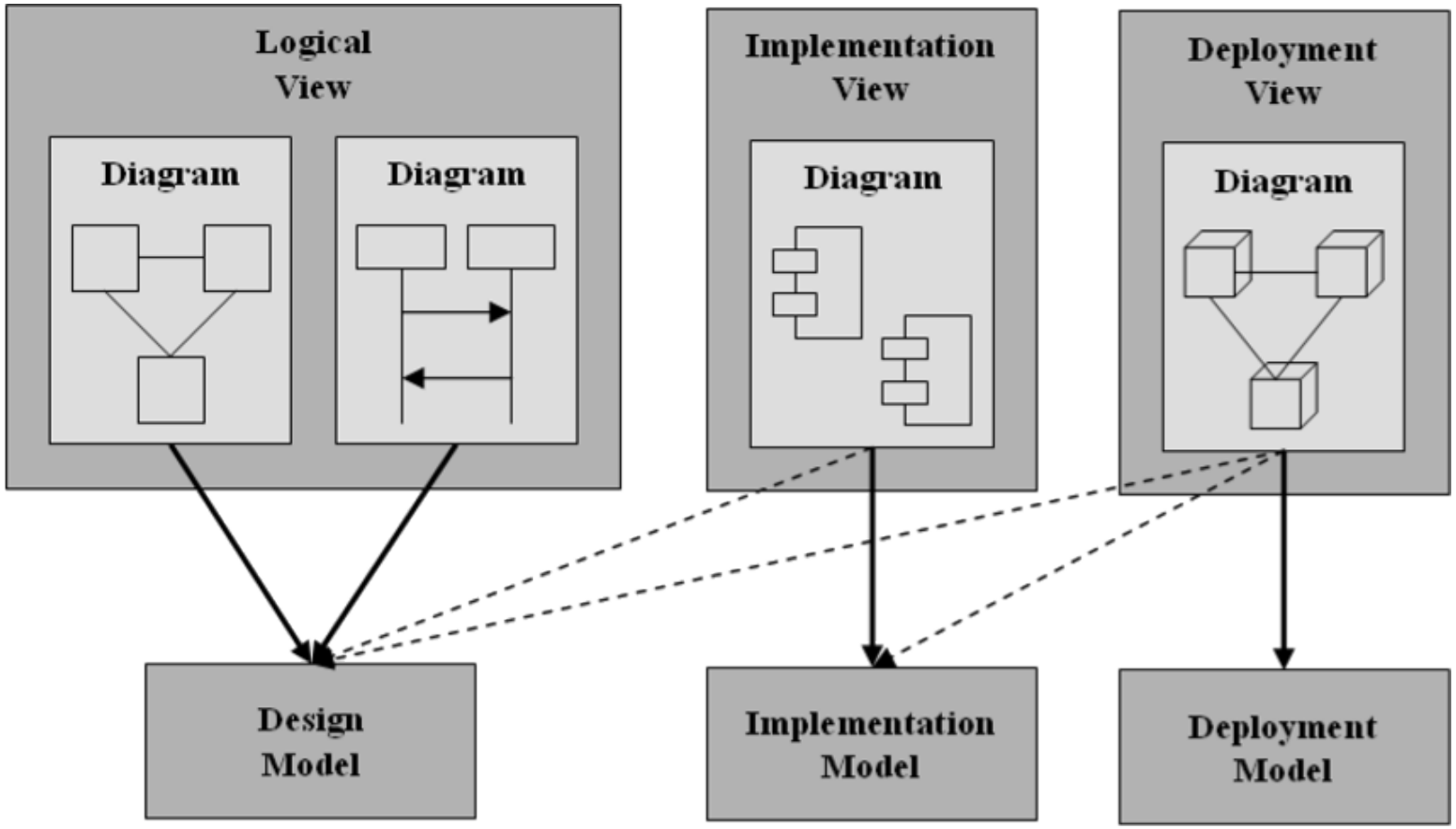
Why is Documentation Important?

- Others can understand and evaluate the design.
- Others in the team can learn from the architecture by digesting the thinking behind the design.
- We can do analysis on the design, perhaps to assess its expected performance.

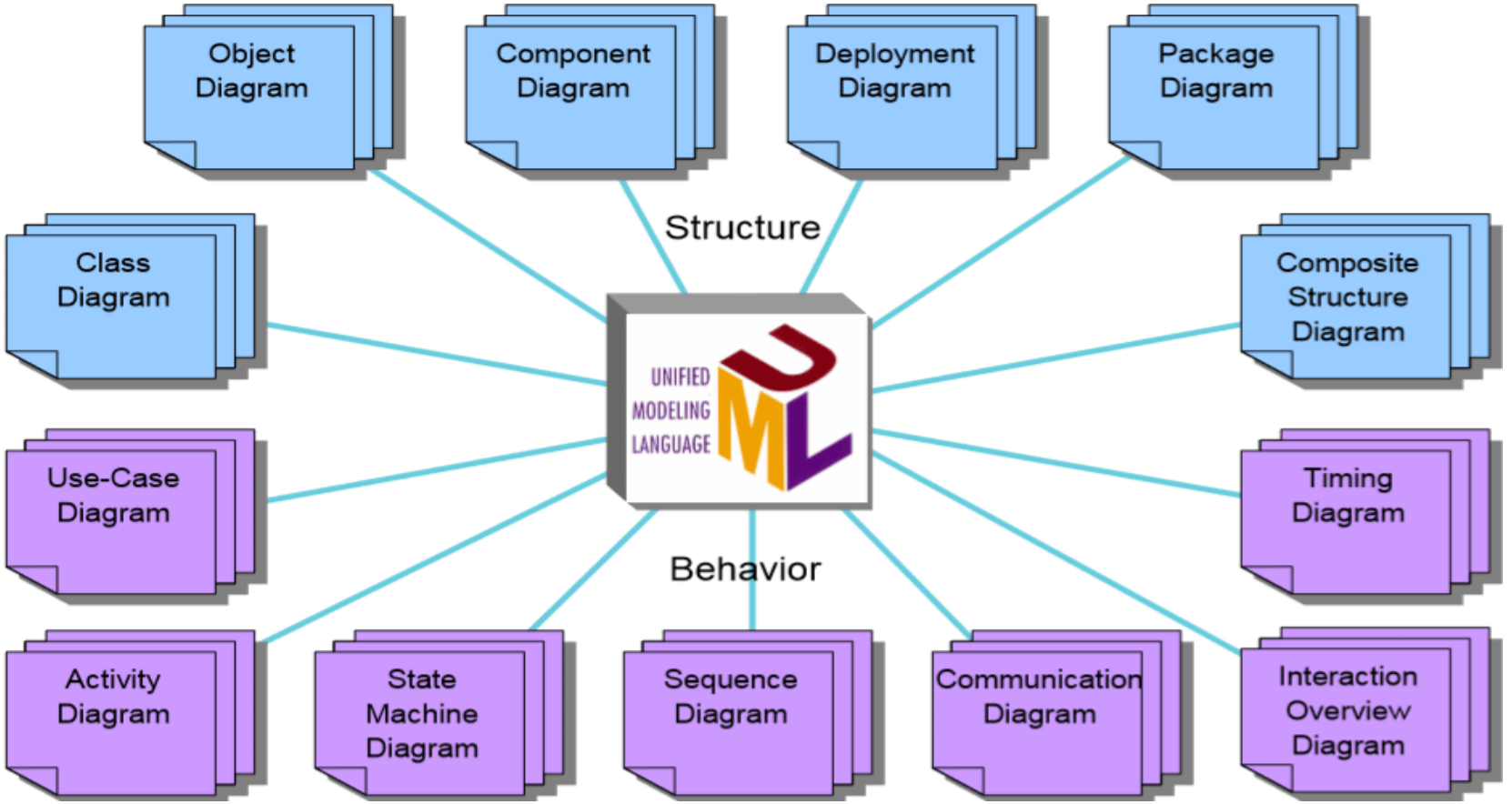
Challenges

- It exists no unique and generally accepted standard.
- Describing complex matters in an understandable way is time consuming and itself complex.
- Architecture is not a static thing

Views, Diagrams and Models



Architectural Documentation Using UML



Design Patterns (dt. Entwurfsmuster)

What is a Design Pattern?

- general, reusable solution to a commonly occurring problem in software design
- description or template for how to solve a problem
- formalized best practices that developers can use to solve common problems
- defines a vocabulary
- most famous are GoF (Gang of Four) patterns described in "Design Patterns: Elements of Reusable Object-Oriented Software", 1994
  - creational patterns
  - structural patterns
  - behavioral patterns

Pattern Documentation (used by GoF)

- Pattern Name and Classification
- Intent
- Also Known As
- Motivation (Forces)
- Applicability
- Structure
- Participants
- Collaboration
- Consequences
- Implementation
- Sample Code
- Known Uses
- Related Patterns

Creational Patterns

- Abstract Factory
- Builder
- Factory Method
- Prototype
- Singleton

FH University of Applied Sciences TECHNIKUM WIEN

Structural Patterns

- Adapter
- Bridge
- Composite
- Decorator
- Facade
- Flyweight
- Proxy

Behavioral Patterns

- Chain of Responsibility
- Command
- Interpreter
- Iterator
- Mediator
- Memento
- Observer
- State
- Strategy
- Template Method
- Visitor

Many More Patterns

- Data Access Object
- Dependency Injection
- Front Controller
- Method Chaining
- Null Object
- Read-Write Lock
- …

Software Components

Software Element

- contains sequences of program statements that describe computations to be performed by machine

Software Component

- software element that conforms to component model and can be independently deployed and composed without modification according to composition standard

Component Model

- defines specific interaction and composition standards
- guideline on how to construct an individual component

Elements of Component Model

- interfaces
- naming
- meta data
- interoperability
- composition
- evolution support
- packaging and deployment

Interface

- abstraction aimed at software reuse
- not a constituent part of component, but serves as contract
- elements
  - operations
  - parameters and their types
  - return values and their types
  - exceptions
  - pre- and post-conditions
  - …

Naming

- uniquely identifiable components and interfaces
  - standardized naming schema
  - hierarchical name spaces

Meta Data

- for composition tools and reflective programs

Interoperability

- communication channels for components
  - inside processes
  - across processes and over the network

Composition

- combination of two or more components

Evolution Support

- components might have to be replaced by newer versions
  - modified and new interfaces
  - co-existence of different versions of one component

Packaging and Deployment

- packaging might include
  - program code (binary vs. source)
  - configuration data
  - other components
  - additional resources
- used for installation and configuration

Component Model Implementation

- set of executable software elements required to support execution of components that conform to model

Interaction Standard

- components may have explicit context dependencies on operating system, software components or some other software elements
- specifies type of explicit context dependencies components may have

Composition Standard

- defines how components can be composed to create larger structure and how producer can substitute one component to replace another that already exists within the structure

Software Component Infrastructure

- set of interacting software components designed to ensure that software system using those components and interfaces will satisfy clearly defined performance specifications

Performance Specification

- defines functional requirements for product, environment in which it must operate, and any interface and interchangeability requirements
- provides criteria for verifying compliance

Component-Based Software Lifecycle

- lifecycle process for software component
  - business rules
  - business process modeling
  - design
  - construction
  - continuous testing
  - deployment
  - evolution
  - subsequent reuse
  - maintenance

Component Platforms

- CORBA (Common Object Request Broker Architecture)
- Microsoft .NET Framework
- Java EE (Java Enterprise Edition)
- …

Web Services

Herausforderungen

- "grüne Wiese" als Wunschvorstellung vs. heterogene Systemlandschaft als Realität → Integration neuer Systemteile komplex
- Interaktion zwischen verschiedenen Systemen steigt stetig
- zukünftige Anforderungen von Systemarchitekten immer schwieriger abschätzbar

Service

- bietet bestimmte Funktionalität
- läuft als eigenständige und modulare Applikation ohne Abhängigkeit zu anderen Diensten
- weist standardisiertes Interface auf
- grobgranular (d. h. Service → Komponente → Modul → Paket → Klasse → Methode)
- geringer Integrationsaufwand
- sollte keinen Status besitzen
- asynchrone und synchrone Funktionsbereitstellung möglich
- erlaubt Komposition
- besitzt messbare QoS (Quality of Service)-Attribute

Eigenschaften von Services

- beschreibbar
- publizierbar
- auffindbar
- bindbar
- aufrufbar
- zusammensetzbar

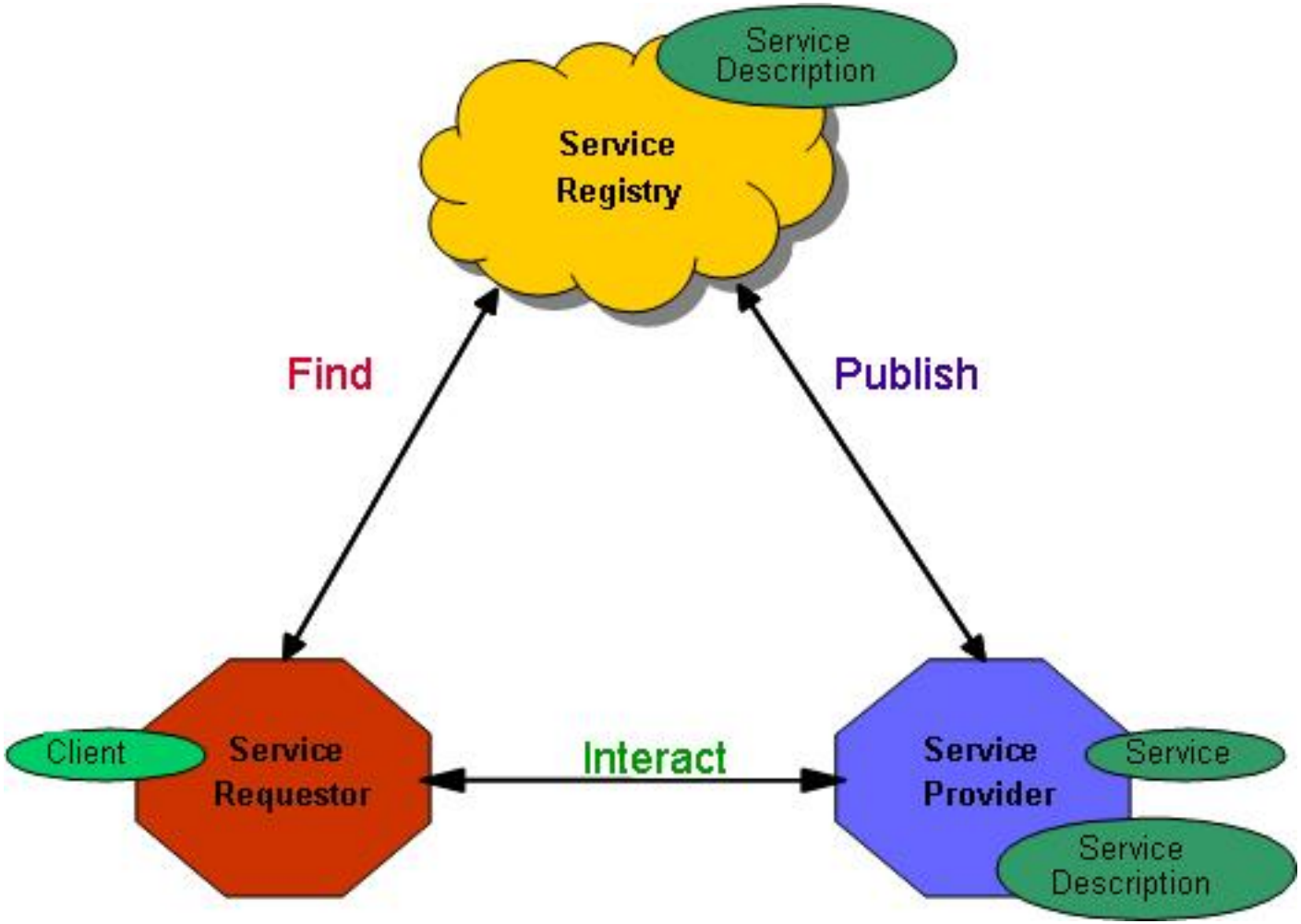
SOA (Service-orientierte Architektur)

- Architekturstil basierend auf Services
- behandelt alle Aspekte von Erzeugung bis Nutzung von Services
- wesentliche Paradigmen
  - lose Kopplung
  - dynamisches Binding
  - hohe Interoperabilität

Rollen bei SOA

- Service Registry
- Service Provider
- Service Requestor

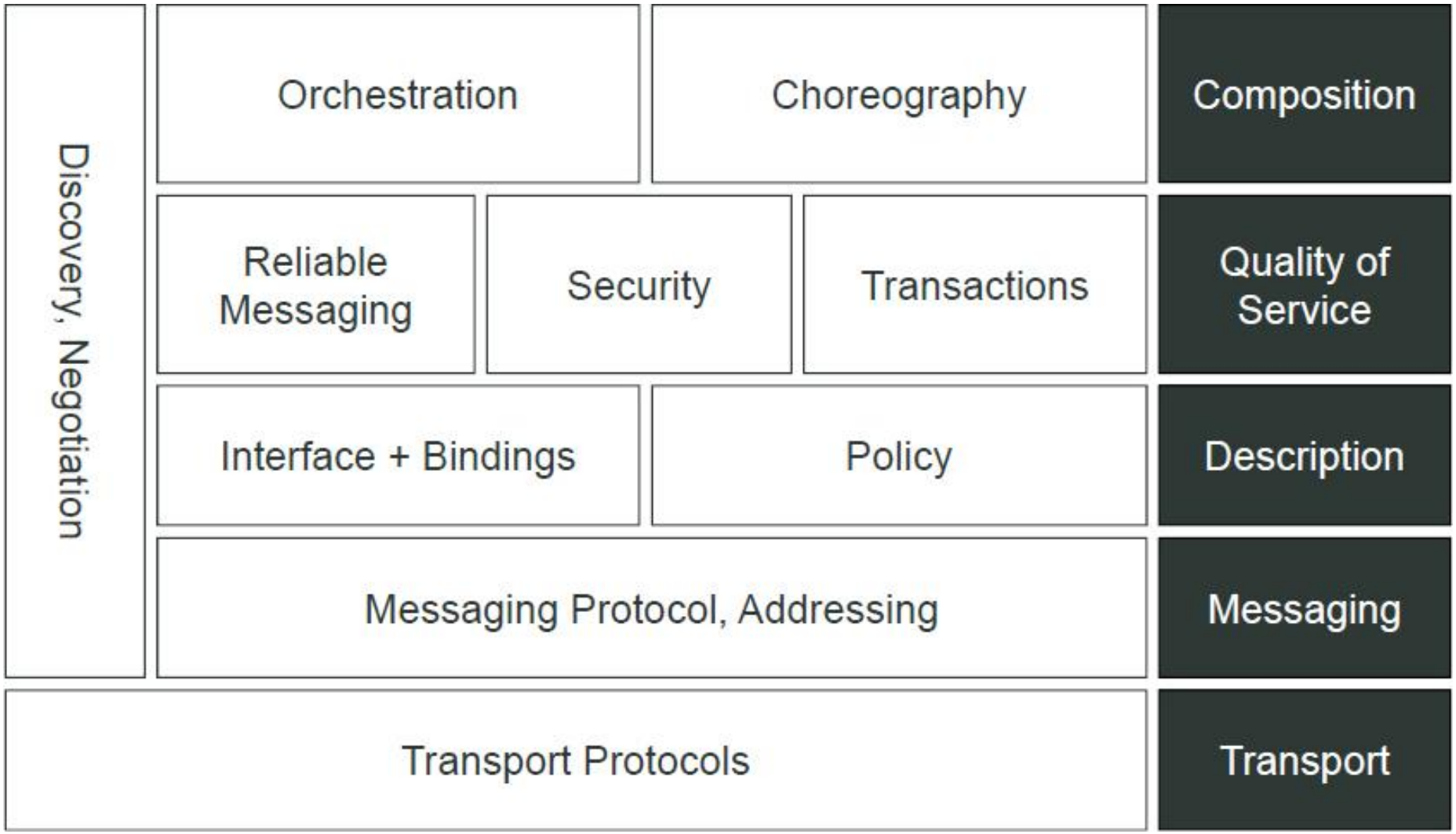
SOA-Triangel



Interface vs. Implementierung

- Interface definiert und beschreibt Funktionalitäten für Aufrufer
  - verfügbare Operationen inkl. Parametern, Datentypen und Rückgabetypen
  - verfügbare Aufruf-Protokolle
- Implementierung realisiert bestimmtes Interface und versteckt Details vor Aufrufer

SOA-Stack


Discovery, Negotiation
Orchestration
Choreography
Composition
Reliable Messaging
Security
Transactions
Quality of Service
Interface + Bindings
Policy
Description
Messaging Protocol, Addressing
Messaging
Transport Protocols
Transport

SOAP Web Services

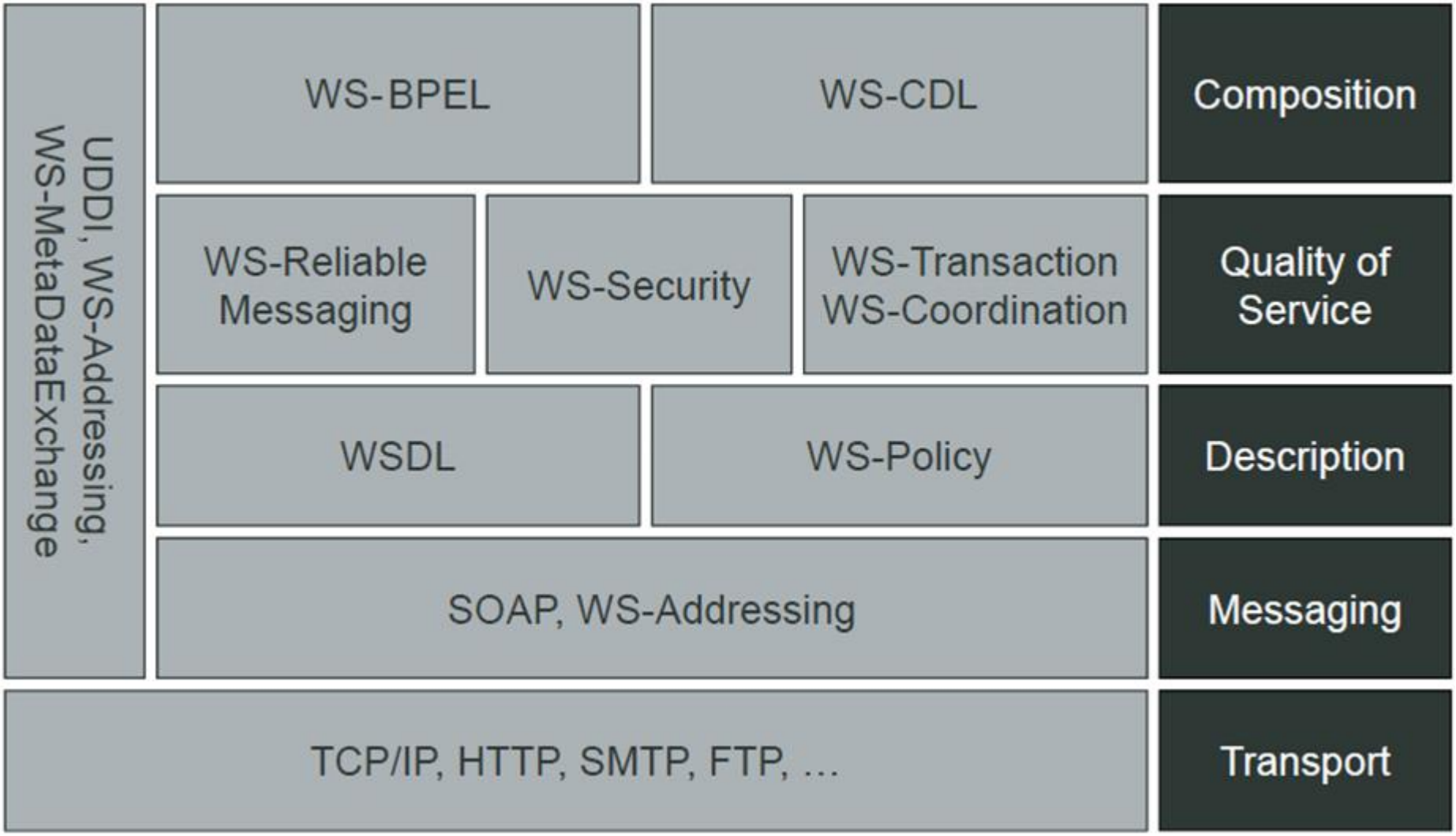
SOAP Web Service

- "A Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP-messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards."

Charakteristik

- auf Web-Standards basierend
  - HTTP als Transportprotokoll
  - XML und XML-Schema zur Datenrepräsentation
- viele Standards und Spezifikationen verfügbar (z.B. von W3C, OASIS)
- plattformunabhängig
- weit verbreitet

WS-Stack


UDDI, WS-Addressing, WS-MetaDataExchange
WS-BPEL
WS-CDL
Composition
WS-Reliable Messaging
WS-Security
WS-Transaction WS-Coordination
Quality of Service
WSDL
WS-Policy
Description
SOAP, WS-Addressing
Messaging
TCP/IP, HTTP, SMTP, FTP, …
Transport

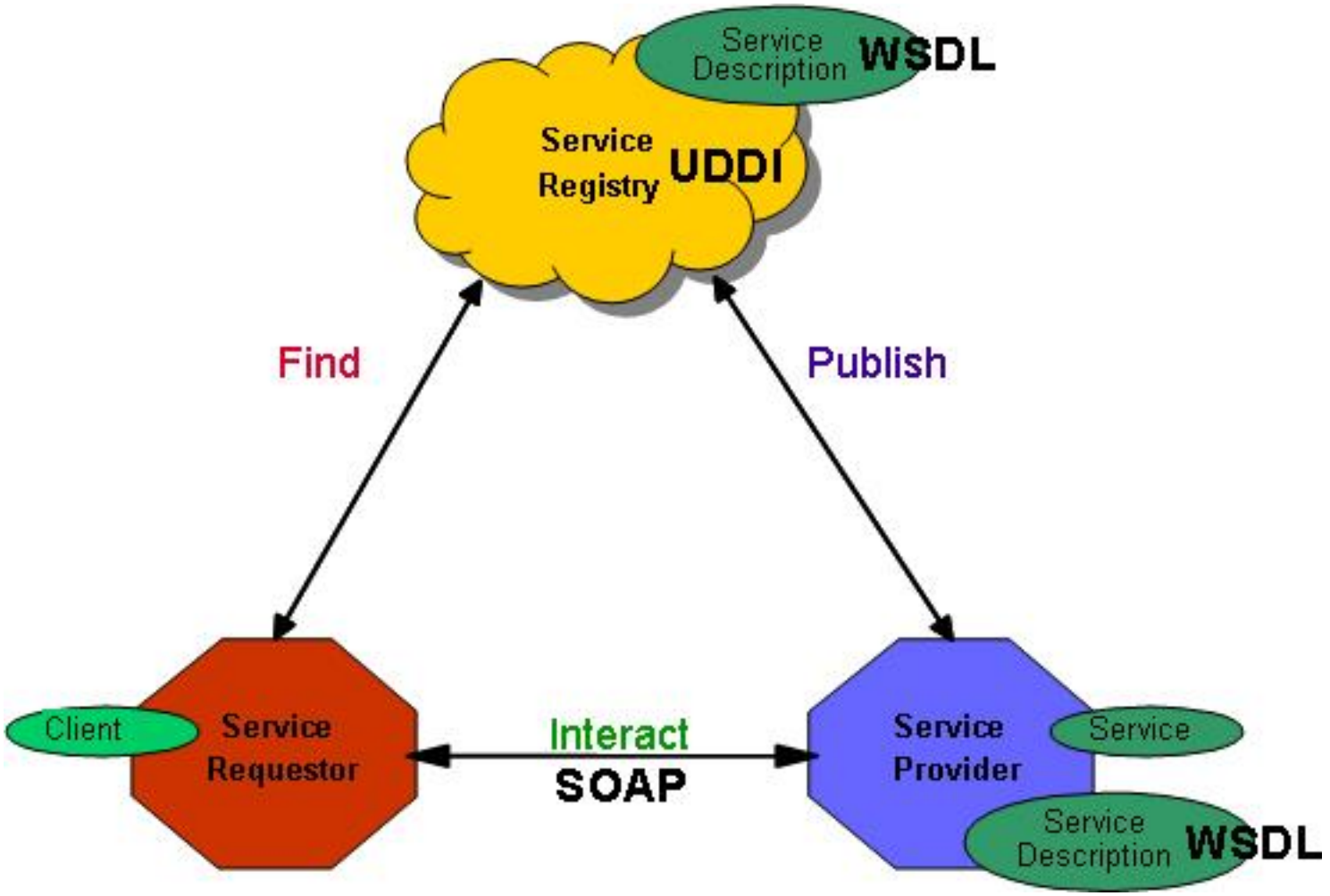
WS-Standards

- SOAP, WSDL und UDDI beschreiben grundlegende Teile der Web Service-Plattform
- Möglichkeit zur Erweiterung von Standards von Haus aus vorgesehen
- erweitere WS-Spezifikationen bieten zusätzliche Features

SOAP, WSDL und UDDI

- SOAP (Simple Object Access Protocol)
  - XML-basiertes Nachrichtenprotokoll für Aufruf von Web Services
- WSDL (Web Services Description Language)
  - XML-basiertes Vokabular zur Beschreibung von Web Services
  - aus zwei Teilen bestehend
    - abstrakt: Operationsverhalten (was?)
    - konkret: Binding (wie?) und Dienst (wo?)
- UDDI (Universal Description, Discovery and Integration)
  - flexibler Verzeichnis- und Registrierungsdienst für Web Services
  - "gelbes Telefonbuch" für Web Services

WS-Triangel


Service Description
WSDL
Service Registry
UDDI
Find
Publish
Client
Service Requestor
Interact
SOAP
Service Provider
Service
Service Description
WSDL

Erweitere WS-Spezifikationen

- WS-Addressing
- WS-BPEL
- WS-BusinessActivity
- WS-Policy
- WS-ReliableMessaging
- WS-Security
- WS-Transaction
- …

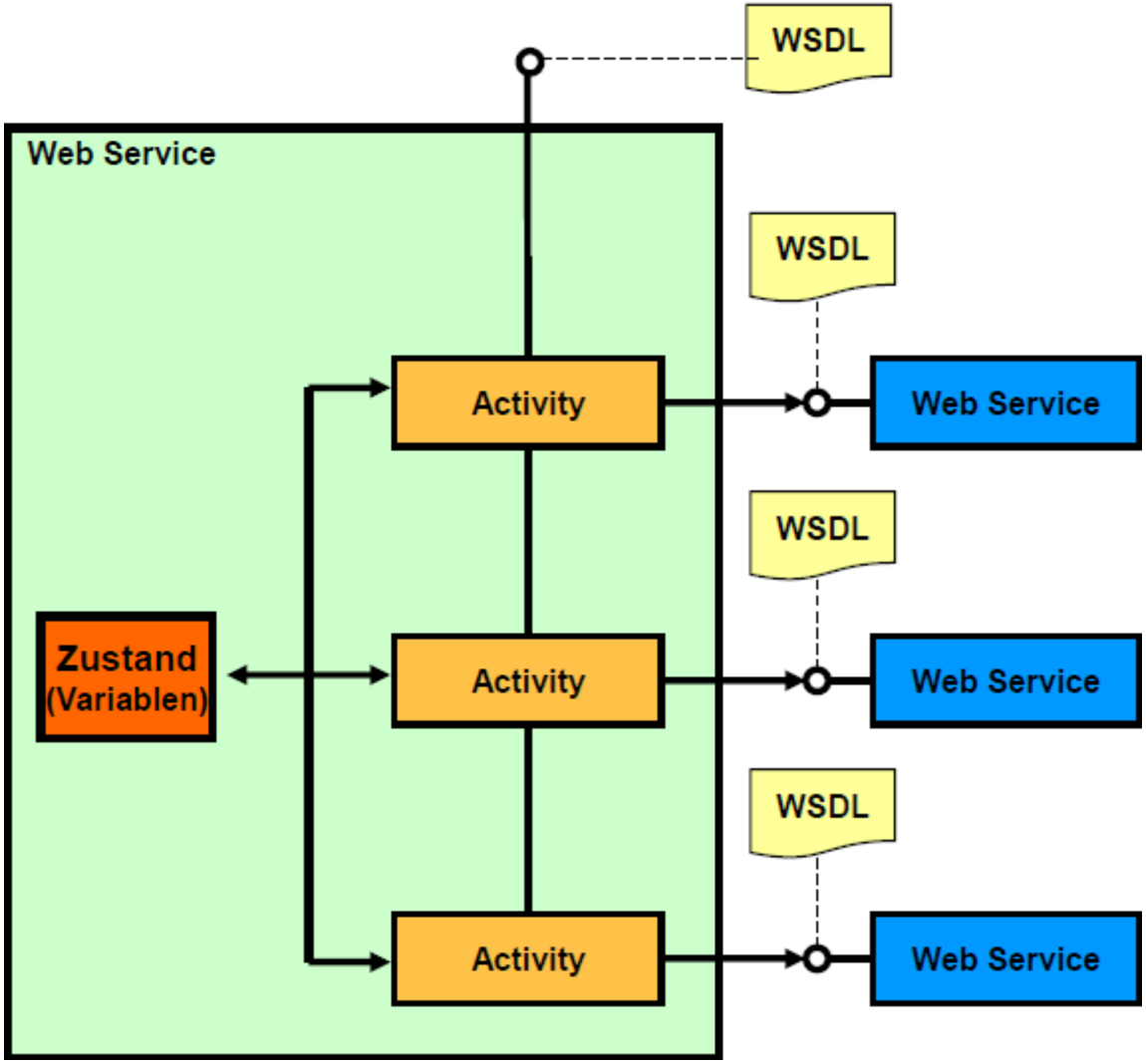
Eigenschaften von SOAP Web Services

- beschreibbar: WSDL
- publizierbar: UDDI
- auffindbar: UDDI
- bindbar: SOAP
- aufrufbar: SOAP
- zusammensetzbar: WS-BPEL, WS-CDL

WS-BPEL

- von IBM, BEA Systems, Microsoft und SAP im Jahr 2002 eingeführt (vormals BPEL4WS)
- XML-basierte Sprache zur Beschreibung von Business-prozessen, deren Aktivitäten durch Web Services implementiert sind, d. h. ermöglicht Orchestrierung von Web Services
- WS-BPEL macht Programmieren im Großen möglich
- durch WS-BPEL beschriebene Businessprozesse stehen selbst wiederum als Web Services zur Verfügung
- BPEL-Server oder BPEL-Engine für Ausführung der durch WS-BPEL beschriebenen Businessprozesse erforderlich

Businessprozess mit WS-BPEL



RESTful Web Services

REST (Representational State Transfer)

- Architekturstil für verteilte Systeme
- Architektur von World Wide Web basiert darauf

Designziele

- Skalierbarkeit
- Robustheit
- Einfachheit der Interfaces
- Unabhängigkeit der Komponenten

Charakteristik

- Resource-Orientation
  - Ressourcen als Schlüsselelemente jedes RESTful Systems
- Statelessness
  - jeder Request eigenständig
- Uniform Interface
  - Zugriff auf jede Ressource über dieselbe Schnittstelle
- Naming
  - jede Ressource hat eindeutigen Namen
- Layering
  - transparentes Dazwischenschalten von Vermittlern möglich

Hauptprotokolle

- HTTP als Kommunikationsprotokoll
- URI zur Identifizierung von Ressourcen

Technische Charakteristik

- Resource-Orientation
  - Repräsentation von Ressourcen durch Media-Types (z.B. application/xml, application/json)
- Statelessness
  - statusloses HTTP
- Uniform Interface
  - HTTP-Schnittstelle mit Methoden POST, GET, PUT und DELETE
- Naming
  - jede Ressource über URI eindeutig adressierbar
- Layering
  - HTTP funktioniert über Proxys, Caches, Gateways und Router hinweg

SOAP vs. REST

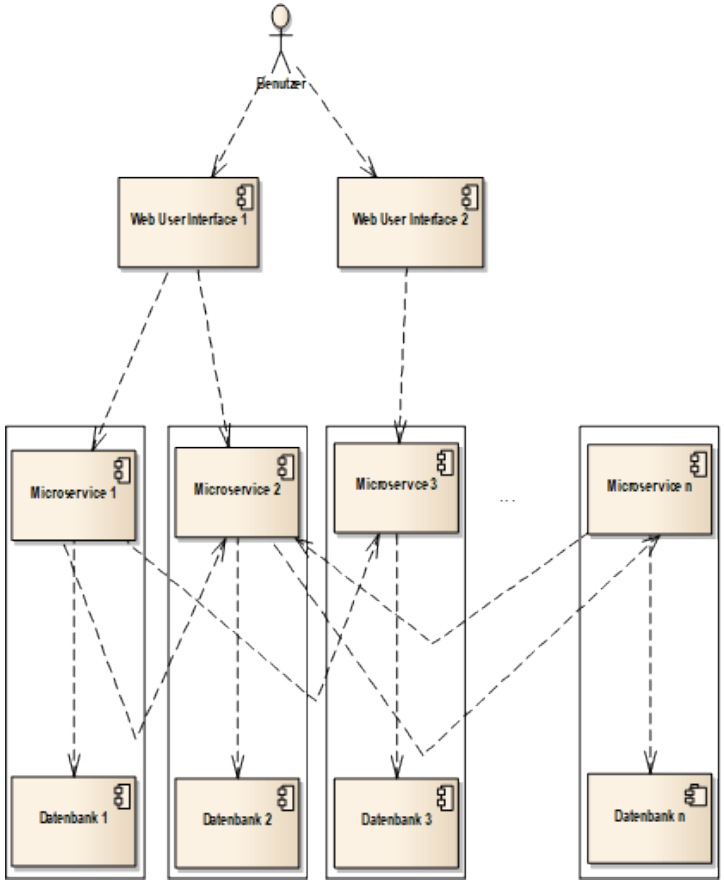
SOAP	REST
Operationen (z.B. Überweisung)	Ressourcen (z.B. Buch)
Hauptfokus auf SOA und EAI	Hauptfokus auf Web 2.0
WS-Stack unterstützt Enterprise-Features	einfache Nutzung und leichtgewichtig
Allgemeingültigkeit: decke möglichst alle Anwendungsfälle durch entsprechende Spezifikationen ab	Einfachheit: halte Standards so einfach wie möglich
Web Services als Wunderwaffe des verteilten Computings	Web Services als einfach nutzbare Web-Integrationstechnologie

Microservices

Microservices

- kleine, eigenständige und miteinander kollabierende Services
- sind hoch spezialisiert und bieten jeweils nur eine stark abgegrenzte Funktionalität an
- implizieren bestimmte Softwarearchitektur

Microservice-Architektur



Vorteile durch Microservices (1)

- Services werden über standardisierte Schnittstellen (z.B. REST) angeboten und sind damit unabhängig von einer bestimmten Technologie.
- Durch getrennte Datenhaltung können spezialisierte Datenbanksysteme zum Einsatz kommen.
- Die Servicelandschaft kann schrittweise auf neue Technologien gehoben werden.
- Beim Ausfall einzelner Services bleiben andere Services verfügbar.
- Services können getrennt voneinander skaliert werden.

Vorteile durch Microservices (2)

- Services können getrennt voneinander ausgerollt werden.
- Kleinere Teams arbeiten an einer kleineren Codebasis, was die Komplexität des Softwareprojektes entschärft.
- Es wird eine strikte Modularität der Software erzwungen.
- Ein Microservice ist leicht durch einen neuen Service ablösbar.

Probleme durch Microservices (1)

- Aufrufe zwischen Microservices sind zeitaufwendig. Neben der Netzwerklatenz wird Rechenzeit für die Konvertierung der Daten in textuelle Darstellung (z.B. JSON) benötigt.
- Microservices sollten funktional strikt voneinander getrennt sein (Shared-Nothing-Ansatz). Das behindert die Wiederverwendung von Komponenten.
- Jede Kommunikation über das Netzwerk ist unzuverlässig. Ein möglicher Ausfall muss vom aufrufenden Microservice kompensiert werden.
- Bei verteilten Services gibt es keine DB-Transaktionen. Die Services müssen selbst darauf achten, dass die Daten konsistent bleiben (z.B. durch kompensierende Aktionen).

Probleme durch Microservices (2)

- Die freie Wahl der Technologie und voneinander unabhängig agierende Teams können zu einem Zoo von technischen Lösungen führen. Das Gesamtsystem kann dann technisch von einzelnen Personen nicht mehr ganzheitlich verstanden werden.
- Ein Refactoring des Codes innerhalb eines Services ist einfach. Umso komplexer ist ein Refactoring über die Service-Grenzen hinweg, falls Funktionalitäten zwischen den Services verschoben werden müssen.
- Deployment und Betriebsprozesse müssen hochgradig automatisiert werden.

Probleme durch Microservices (3)

- Jeder Service sollte in seiner eigenen Umgebung betrieben werden. Es wird eine Vielzahl von Umgebungen benötigt, um die Microservices unabhängig zu betreiben.
- Um Probleme erkennen zu können, muss jeder Microservice ein Monitoring unterstützen.
- Die Komplexität des Betriebs ist nach Ansicht vieler die größte Herausforderung bei Microservice-Architekturen.