Softwarekomponentensysteme

Software Architecture and Components

Bernhard Löwenstein

Software Architecture

FH University of Applied Sciences
TECHNIKUM WIEN

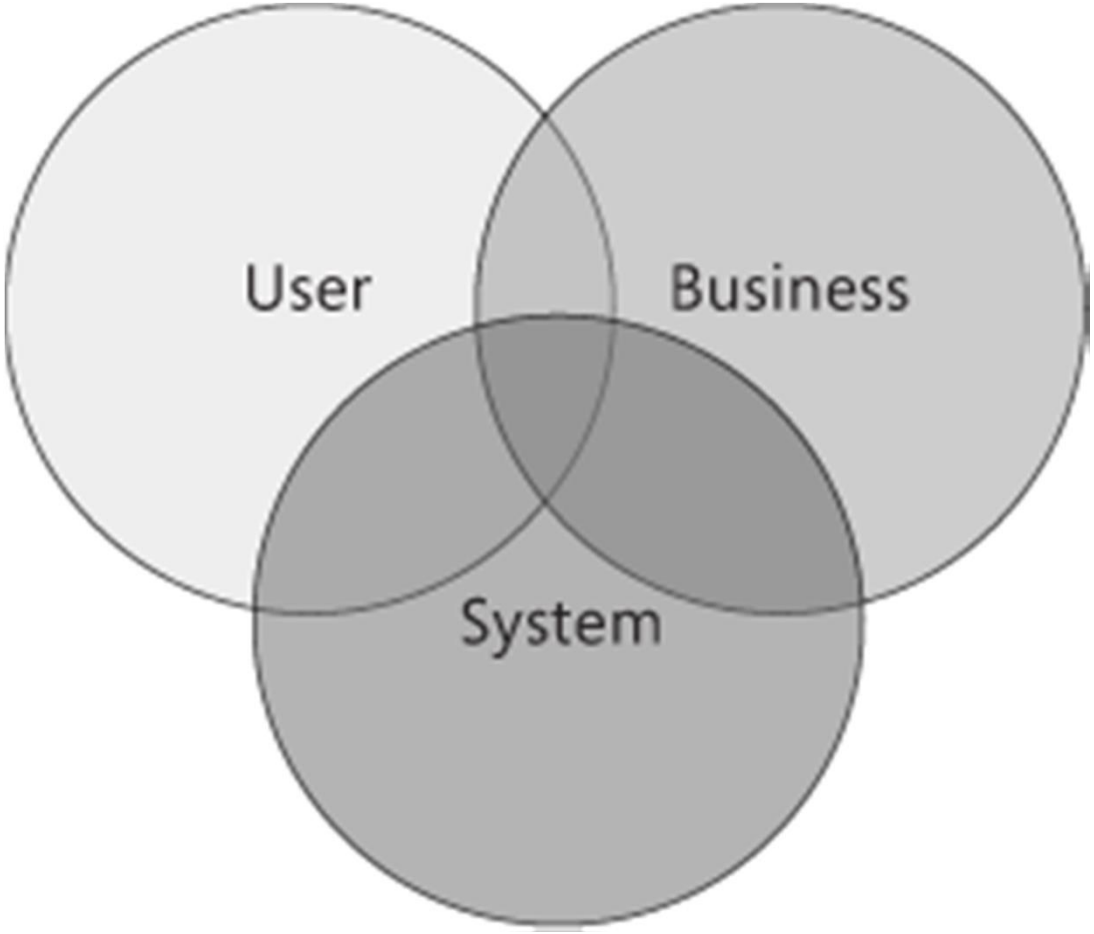
FH University of Applied Sciences TECHNIKUM WIEN

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

FH University of Applied Sciences TECHNIKUM WIEN

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

FH University of Applied Sciences TECHNIKUM WIEN

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

FH University of Applied Sciences
TECHNIKUM WIEN

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

FH University of Applied Sciences TECHNIKUM WIEN

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: CORBA, Jakarta EE, Spring, …

Object-Oriented Architectural Style

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

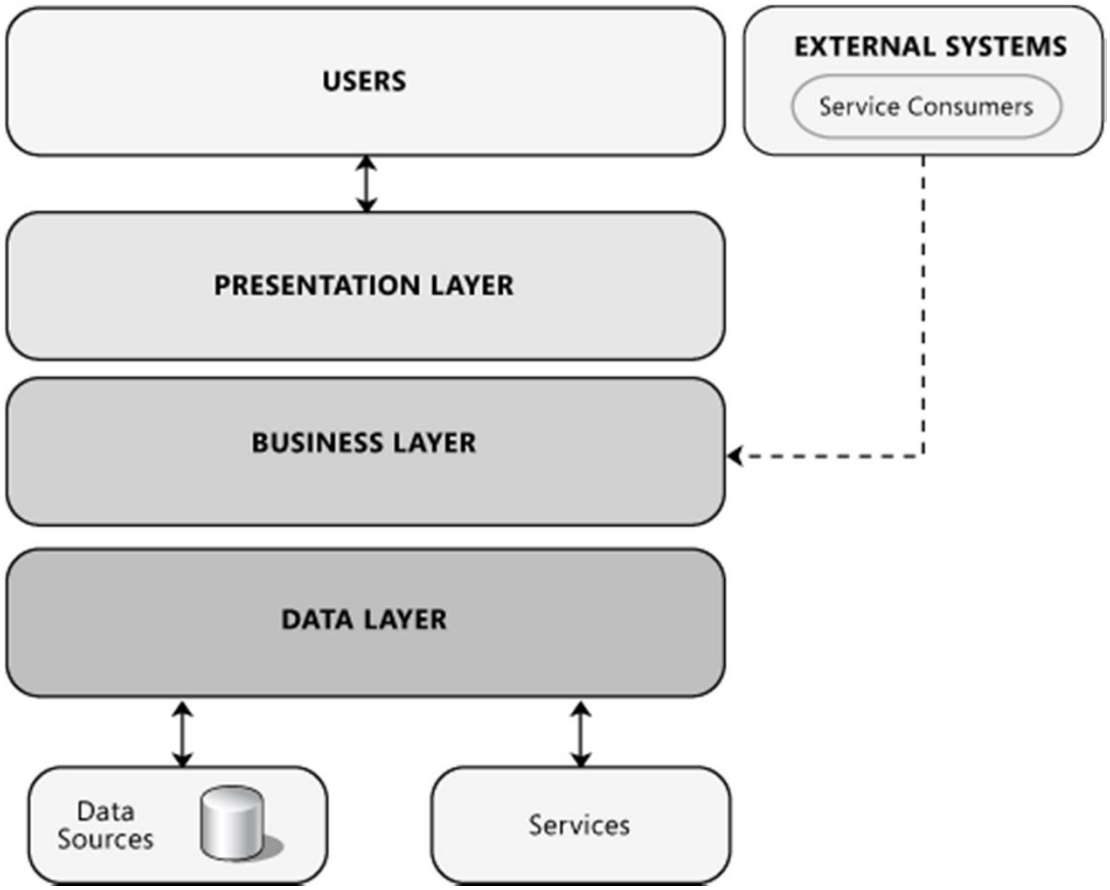
Layered Architectural Style

- 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

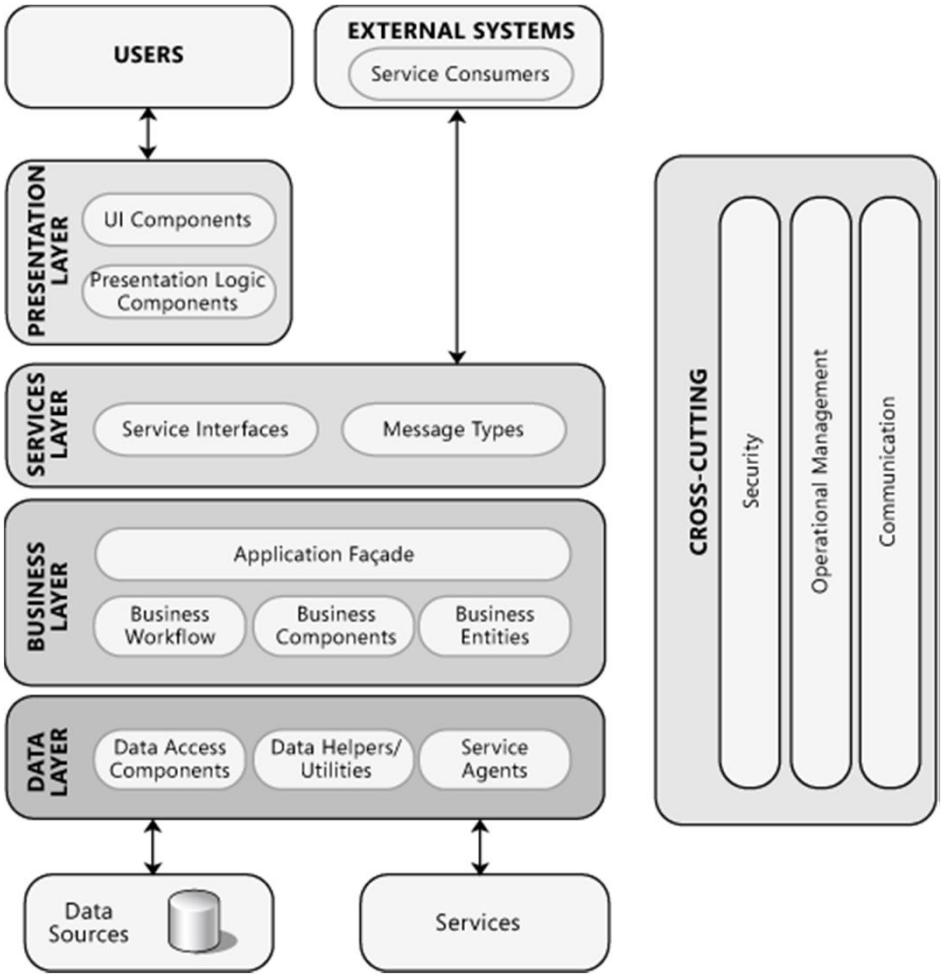
Layered Architectural Style

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

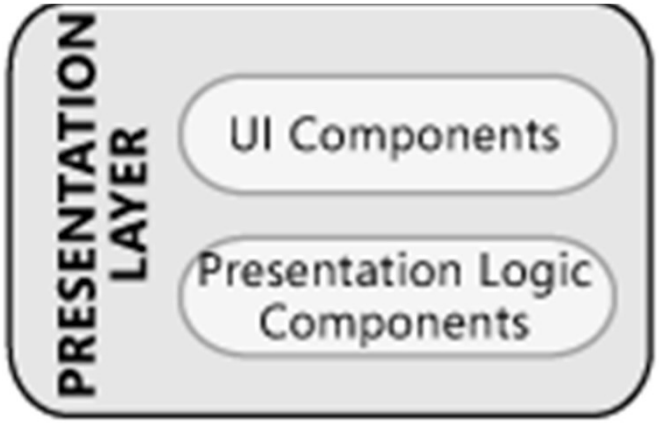
Logical View of 3-Layer Application



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, …)

FH University of Applied Sciences TECHNIKUM WIEN

Presentation Layer Technologies

- Swing
- JavaServer Faces
- JavaFX
- …

FH University of Applied Sciences TECHNIKUM WIEN

Challenges at Presentation Layer

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

Service Layer

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

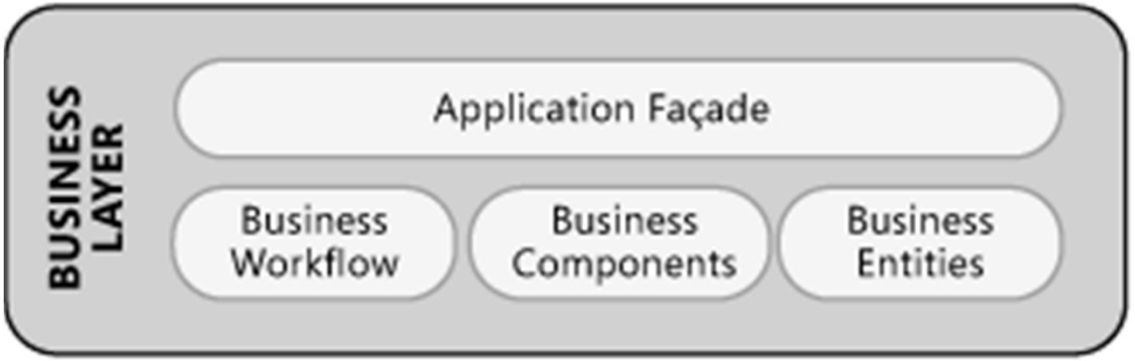
Services Design

- 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

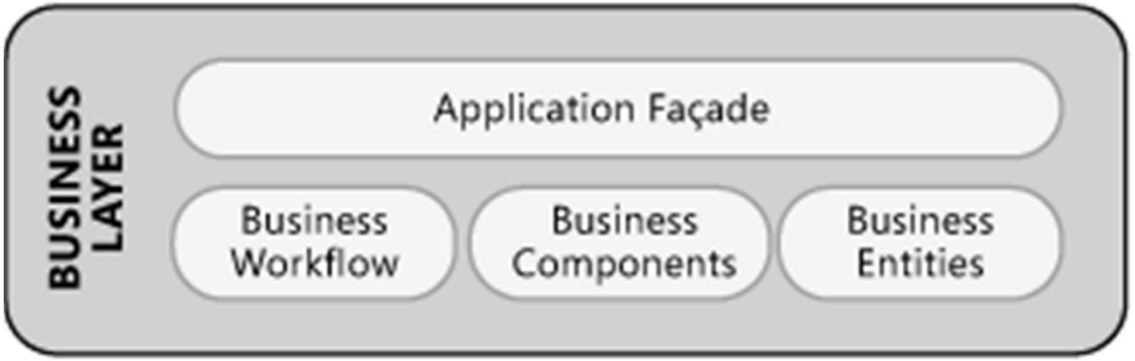
- Protocols:
  - SOAP, REST, …
- Frameworks / Infrastructure:
  - JAX-WS
  - JAX-RS
  - …

Business Layer



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

Business Layer

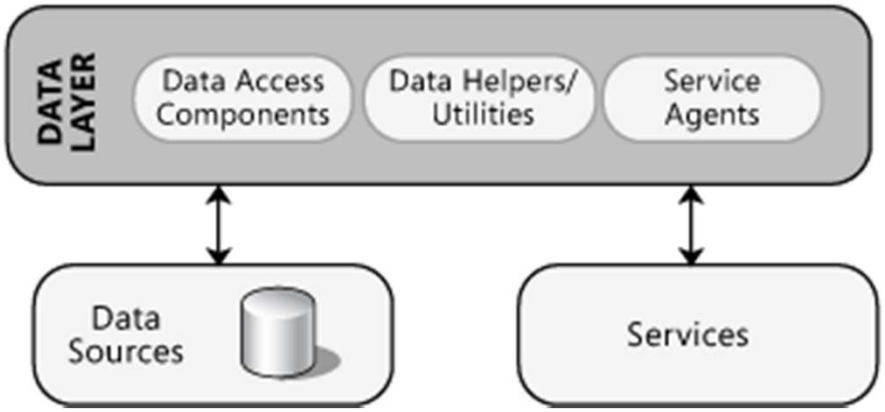


- Business logic components:
  - Business workflow components
  - Business components
  - Business entity components

Business Layer

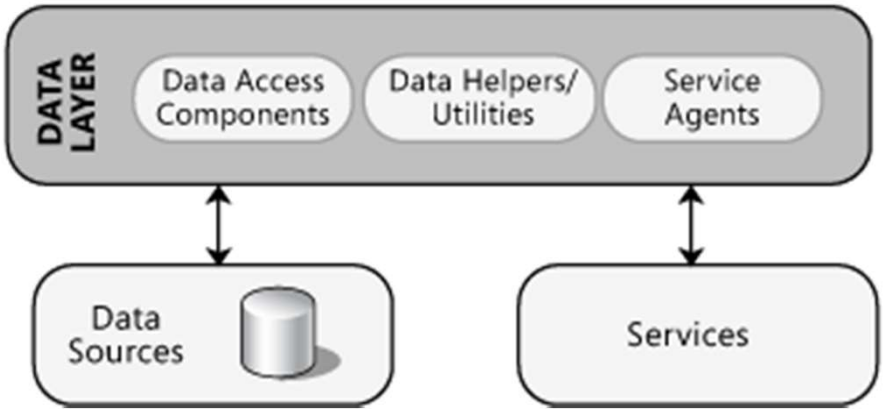
- Business workflow components:
  - Perform business processes
  - Orchestration between multiple steps
  - Long running processes
- Business components
  - Encapsulate logic, calculation, validation, …
- Business entity components
  - Store data (e.g. Customer, Order)
  - Sometimes also: validation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

Data Layer



- Access data sources of different kinds:
  - Relational
  - NoSQL
  - 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

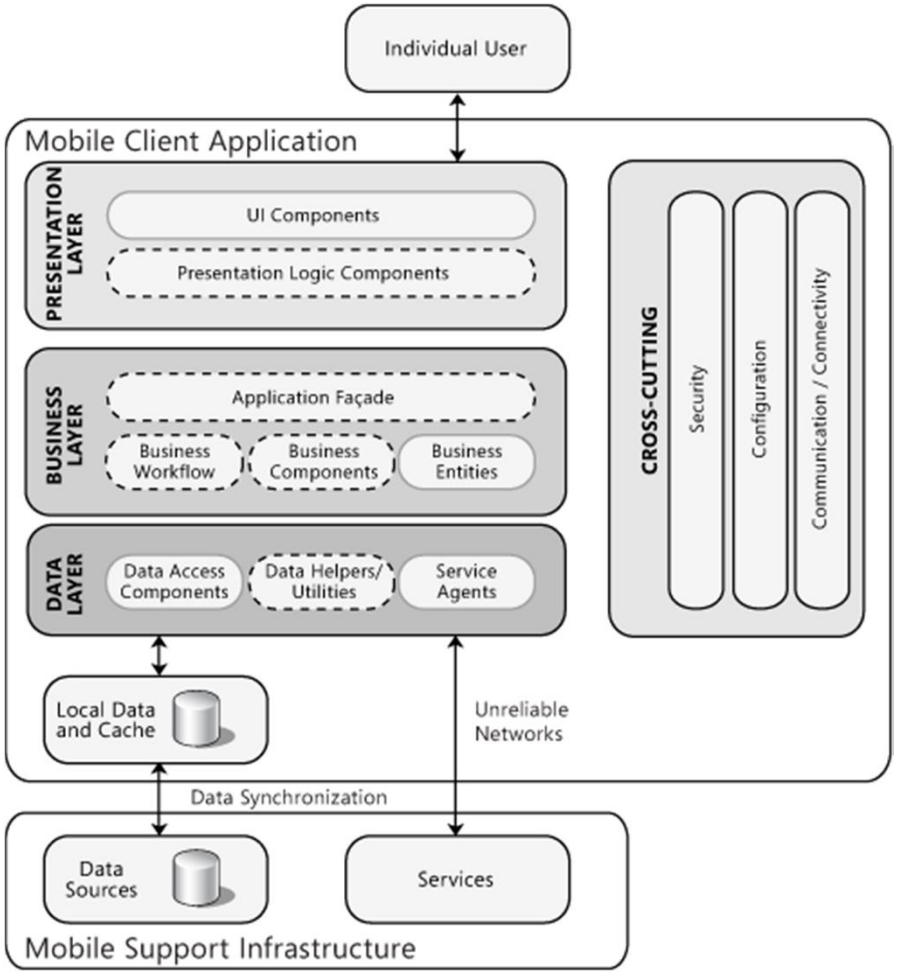
Application Archetypes

FH University of Applied Sciences
TECHNIKUM WIEN

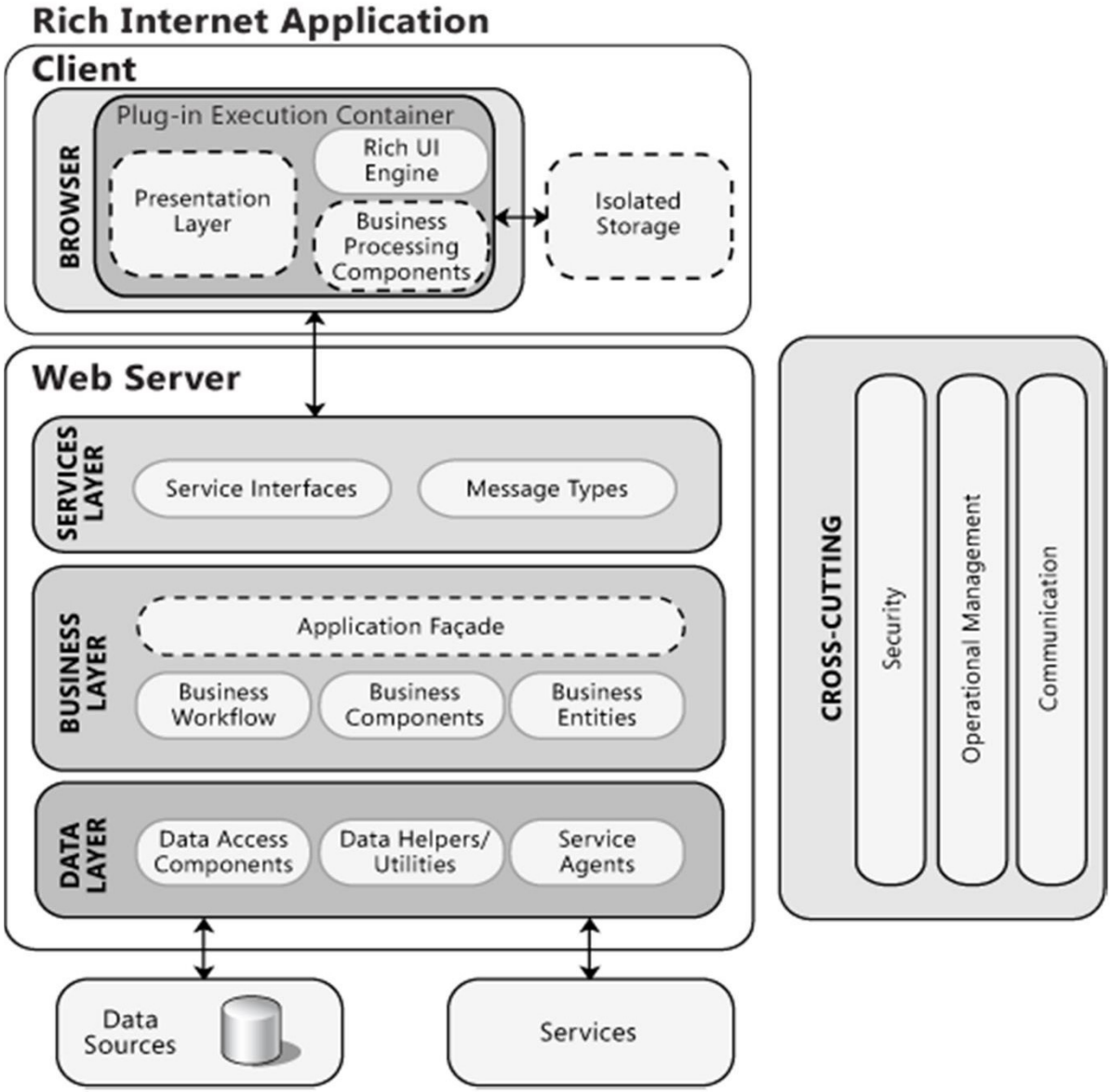
Common Application Archetypes

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

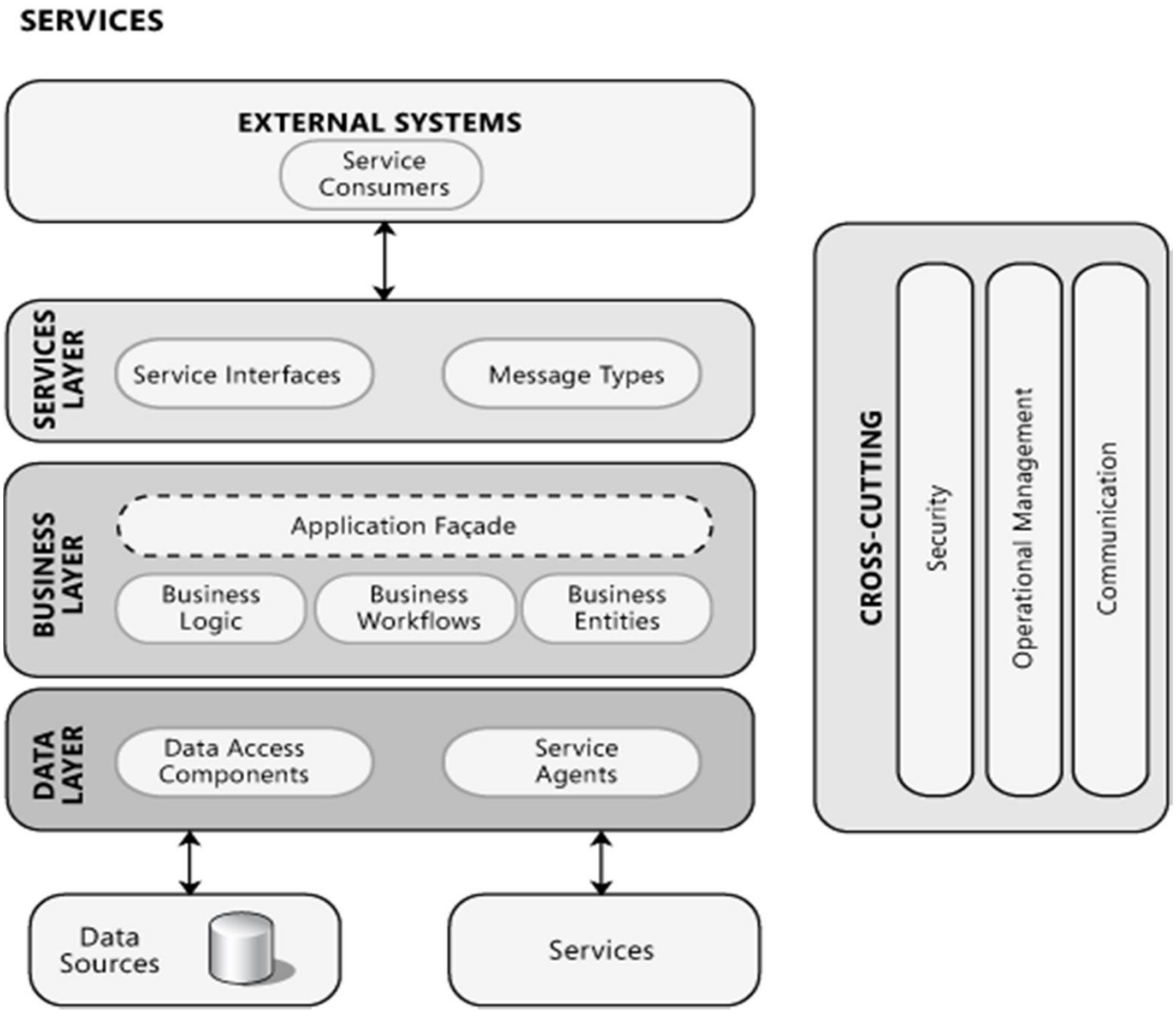
Mobile Application



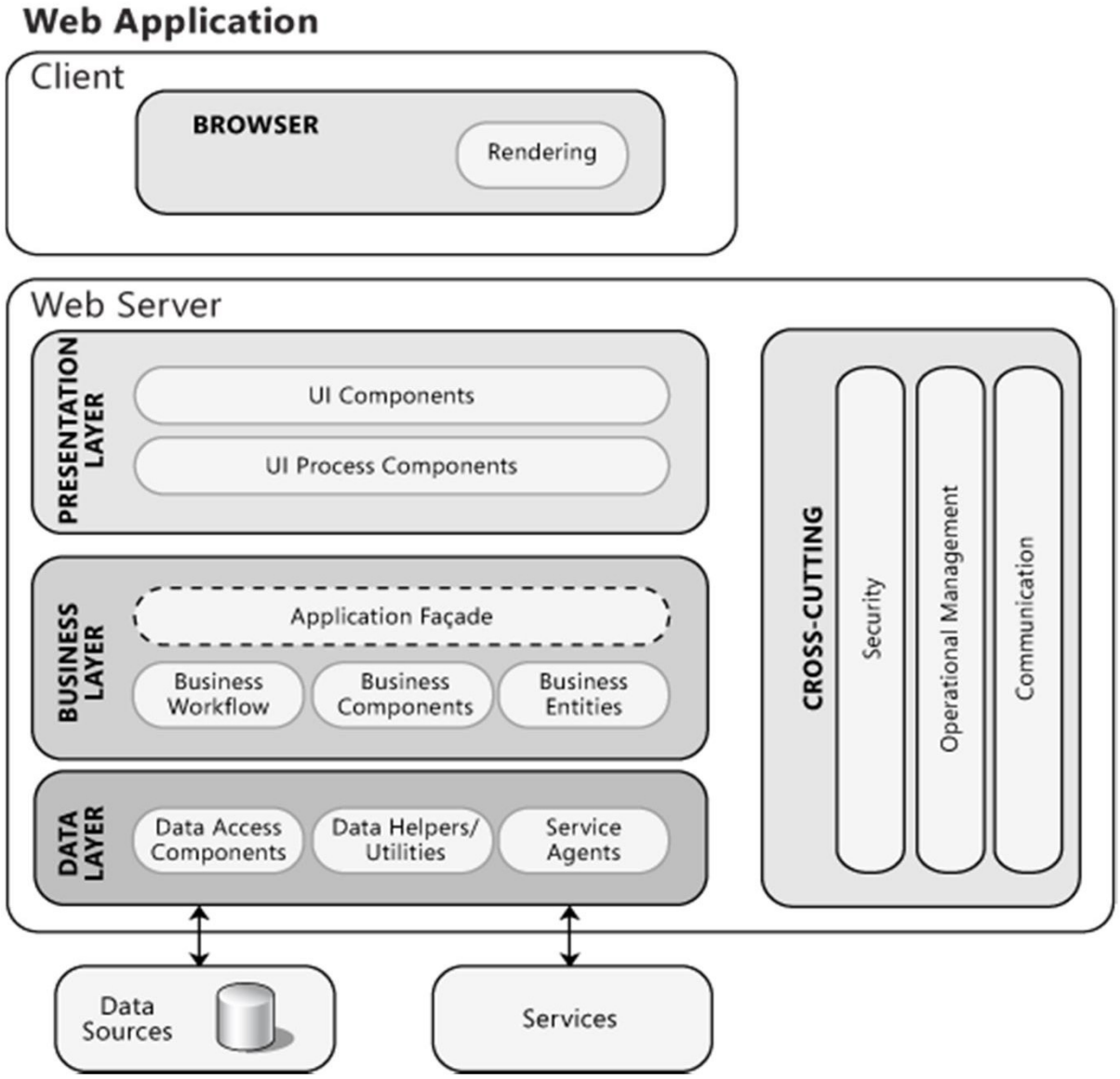
Rich Internet Application



Service Application



Web Application



Architectural Documentation

FH University of Applied Sciences TECHNIKUM WIEN

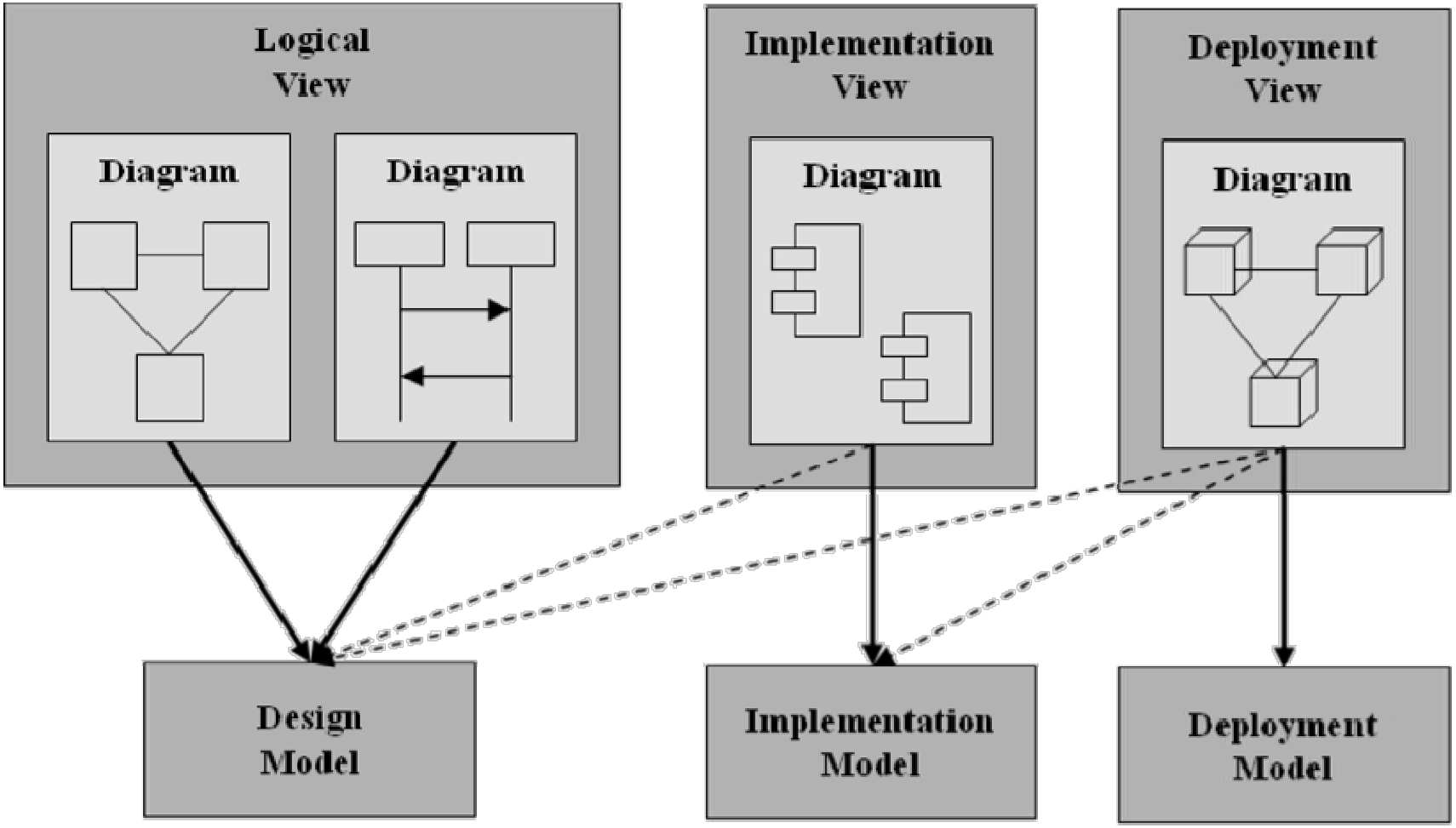
Why is Documentation Important?

- 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.

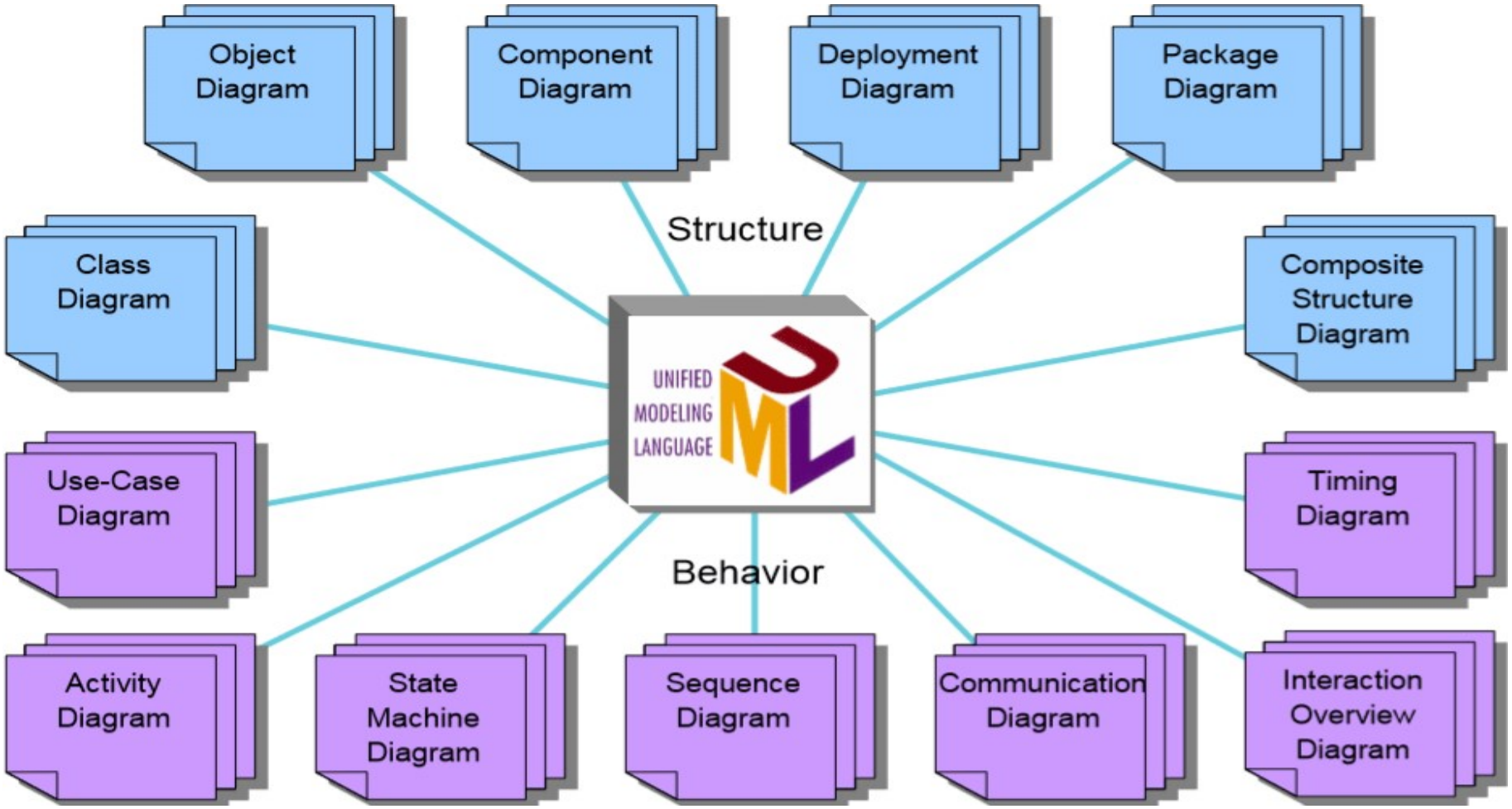
Challenges

- 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)

FH University of Applied Sciences TECHNIKUM WIEN

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 Gang of Four (GoF) 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

FH University of Applied Sciences TECHNIKUM WIEN

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

FH University of Applied Sciences TECHNIKUM WIEN

Behavioral Patterns

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

FH University of Applied Sciences TECHNIKUM WIEN

Many More Patterns

- Data Access Object (DAO)
- Dependency Injection (DI) or Inversion of Control (IoC)
- Front Controller
- Method Chaining
- Null Object
- Read-Write Lock
- …

Software Components

FH University of Applied Sciences
TECHNIKUM WIEN

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

FH University of Applied Sciences TECHNIKUM WIEN

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

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