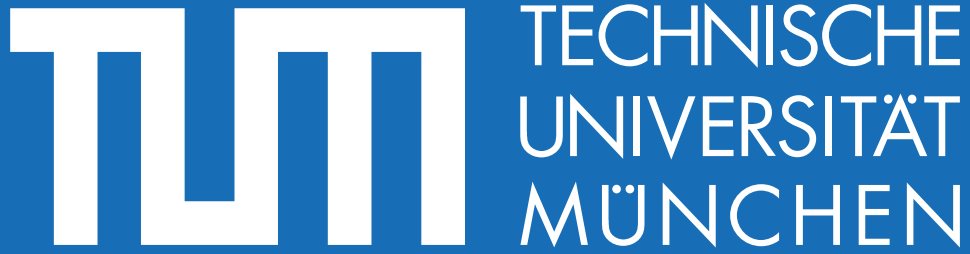



# Software Engineering in der industriellen Praxis (SEIP)

Dr. Ralf S. Engelschall

ARCHITECTURE FUNDAMENTALS

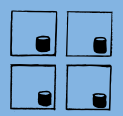
# Architecture Principles

TECHNISCHE UNIVERSITÄT MÜNCHEN

AF 04.2

## FL Factual Locality

Resources are as spatially and temporarily local-scoped to solution components as possible

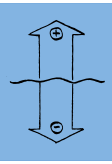


## ES Exclusive Sovereignty

Exclusive resource sovereignty by the enclosing component

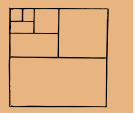
## CA Contextual Adequacy

Neither insufficient nor exaggerated solutions for each context



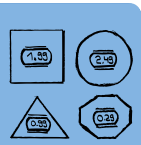
## SP Solution-oriented Proportionality

Good expected proportionality in each solution context



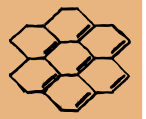
## HC Holistic Consistency

Full consistency across all aspects of a solution



## SH Structural Homogeneity

Maximum homogeneity in the structure of a solution



## CR Constructional Reusability

High reuse of proven structural components and partial solutions


## FS Fulfilled Standards

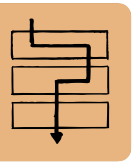
Compliancy to standards as much as possible, as long as the benefits predominate the drawbacks

## FA Functional Abstraction

Suitable level of abstraction across all functional aspects of a solution

## FT Functional Traceability

Suitable traceability across all functional aspects of a solution

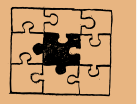


## CI Communicative Interoperability

Maximum interoperability in communication between solutions

## EH Environmental Harmony

Maximum harmony in the integration of the solution with its environment



## AR Avoided Redundancy

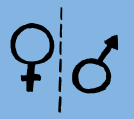
Minimum total number of copies of a single resource

## MS Minimum Special-Cases

Minimum total number of special-cases in a solution

## LS Logical Separation

Separation of concerns between the components of a solution



## SM Structural Modularity

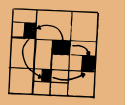
Splitting of a solution into manageable structural components

## LC Loose Coupling

Loose coupling in communication and referencing between solution components

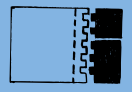
## SC Strong Cohesion

Strong relationship between functionalities within a single solution component



## OE Open Extensibility

Solution components can be extended by third-parties at fixed interfaces



## CC Closed Changeability

Solution components are protected against direct change by third-parties


## UI Unique Identification

Unique identification of all components of a solution


## UA Uniform Addressing

Uniform addressing of all components of a solution


## OS Overall Simplicity

All design aspects of a solution are as simple as possible and only as complicated as necessary

## EC Encapsulated Complexity

Complex related aspects of a solution are encapsulated into a single responsible component

## LA Least Astonishment

All design aspects of a solution are as little astonishing as possible and only as esoteric as necessary

## SD Self Documentation

All design aspects of a solution are preferably self-documenting

## OD Operational Delight

The solution provides users true delight even on long-term operation

## AA Artistic Aesthetics

The solution has holistic aesthetics and artistic love in details

ARCHITECTURE FUNDAMENTALS

# Component Design

TECHNISCHE UNIVERSITÄT MÜNCHEN

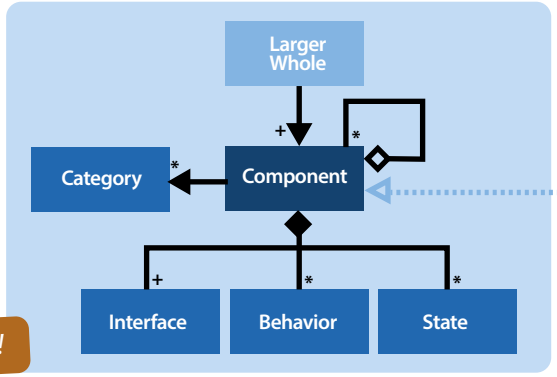
AF 05.1

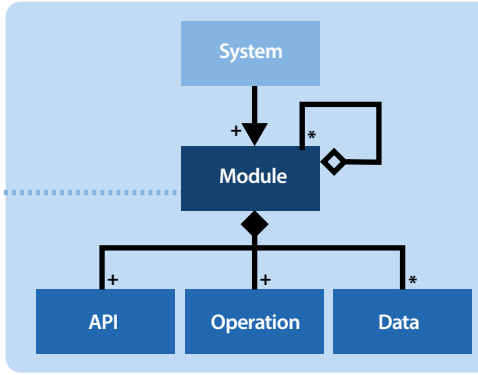
**Definition of a Component (of a Larger Whole):**
a **know-how encapsulating**, potentially **reusable** and **substitutable** **unit of hierarchical composition** with **explicit context dependencies**, which **hides the complexity** of its optional **behavior** and **state** realization behind **small contractually specified interfaces**, defines its **added value** in terms of **provided** and **consumed** interfaces and optionally belongs to zero or more **categories** of similar units.

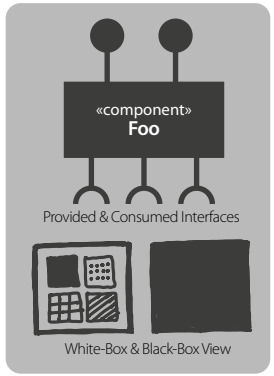
**Definition of a Module (of a System):**
a know-how encapsulating, potentially reusable and substitutable **source-code** unit of hierarchical composition with explicit context dependencies which hides the complexity of its **operation** and **data implementation** behind small contractually specified **Application Programming Interfaces (API)**, defines its added value in terms of provided and consumed **APIs** and optionally belongs to zero or more categories of similar units.

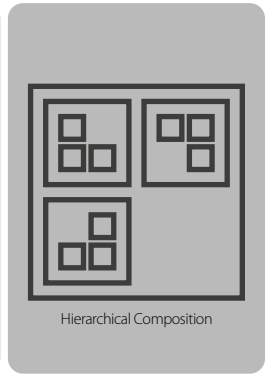
**Example Categories of Components:**
- Namespace
- Directory, File
- Configuration, Section, Directive
- Host, Virtual Machine, Container
- Process Group, Process, Thread
- Application, Microservice, Program
- Package, Class, Function
- Database, Schema, Table, Record
- Datamodel, Entity Group, Entity
- User Interface, Dialog, Widget

*Any group of anything!*



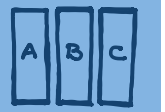

## How to find Components (or Modules)?

### DCA Domain Concept Abstraction
Model domain concepts as entity components and then group at higher levels.

### SOC Separation of Concerns
Build components for clearly distinct concerns.

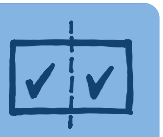
### USE Reusability Potential
Decide on components based on their reusability potential.


### UCC Use-Case Clustering
Build domain components for each use-case or each logical use-case cluster.

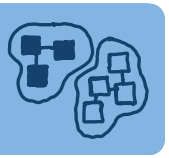



### SRP Single Responsibility Principle
Build components for clearly distinct responsibilities.

### DCC Divide & Conquer Complexity
Master overall complexity by splitting larger things into smaller things.



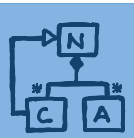
### DDD Domain-Driven Design
Model domain "Bounded Contexts" through DDD and derive components from them.



### CNC Coupling and Cohesion
Decide on components based on their loose coupling and strong cohesion.

### CCC Cross-Cutting Concerns
Build common cross-cutting concerns as cross-cutting components.


### OOD Object-Oriented Design
Model Object-Oriented Design entities (and/or OOP constructs) as components.




### DEP Dependency Encapsulation
Decide on components based on their encapsulation of dependencies.

### PAT Architecture Patterns
Build inner components to comply to outer structure, slicing and clustering architectures.

ARCHITECTURE FUNDAMENTALS

# Interface Design

TECHNISCHE UNIVERSITÄT MÜNCHEN

**Definition of an Interface:**

well-defined shielding and abstracting **boundary** of a passive, providing **component**, consisting of one or more distinguished, **outside-in** designed, **interaction endpoints**, each accessed and controlled by active, consuming components through the **exchange** of **input/output information** and operating under a certain **syntactical** and **semantical contract.**

**Endpoint:** Name, Directive, Command, Function, Method, Procedure, Address, Port, URL, Dialog, ...

**Exchange:** Option, Argument, Parameter, Return Value, Result, Request/Response Message, Error/Exception, Interaction, ...

**Contract:** Syntax, Pre-Condition, Invariant, Post-Condition, Side-Effect, Idempotence, Determinism, Functionality, ...

AF 05.2

## Types of Software Interfaces

### API Application Programming Interface

Example: `foo("bar", 42)` (call and use)


### SPI Service Provider Interface

Example: `register("foo", (x,k) => ...)` (extend and implement)


### SCI Startup Configuration Interface

Examples: INI, Java Properties, TOML, YAML, JSON, XML, etc.

### BPI Batch Processing Interface

Examples: Unix at(1), Unix ts(1), GNU Batch, Spring Batch, Java Batch, SAP LO-BM, etc.

### CLI Command-Line Interface

Example: `foo -x --bar=baz quux`

### GUI Graphical User Interface

Examples: Windows/WPF, macOS/Cocoa, KDE/Qt, GNOME/GTK


### RNI Remote Network Interface

Examples: GraphQL-IO, HTTP/REST, SOAP, RMI, WebSockets, AMQP, MQTT, etc.

## Characteristics of Good Interfaces

### AP Appropriate & Proportional

Appropriate to consumer requirements, proportional to provider functionality.

### SA Shielding & Abstracting

Shields from direct access, abstracts and hides implementation details.

### IE Inviting & Expressive

Invites through "outside – in" design, powerful in expressiveness.

### IF Intuitive & Foolproof

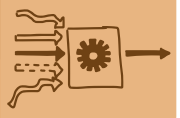
Intuitive to grasp and use, hard to misuse.

### OC Orthogonal & Concise

Supports combinatorial use-cases, causes minimum boilerplate.

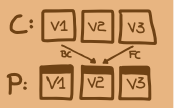
### TP Tolerant & Predictable

Tolerant on input, predictable on output.



### EC Extensible & Compatible

Easy to extend for providers, backward/forward-compatible for consumers.




## Selected Interface Design Patterns

### IVF Interface Version & Features

Provide version and feature information for algebraic comparison and feature detection.


### 2LF Leaky Two-Layer Facade

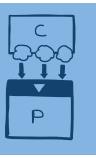
Provide higher-level convenient use-case and lower-level orthogonal feature interface.

### EVE Event Emitter

Emit events to previously registered, interested consumers.

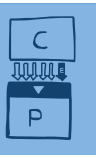
### CTX Multi-Context

Use contexts to distinguish between different usage scenarios and to carry common info.




### CEF Configure-Execute Flow

Spread use-cases onto a flow of configuration exchanges and a final executional exchange.




### IOC Inversion Of Control

Invert control on asynchronous operations via callbacks, promises or async. mechanisms.

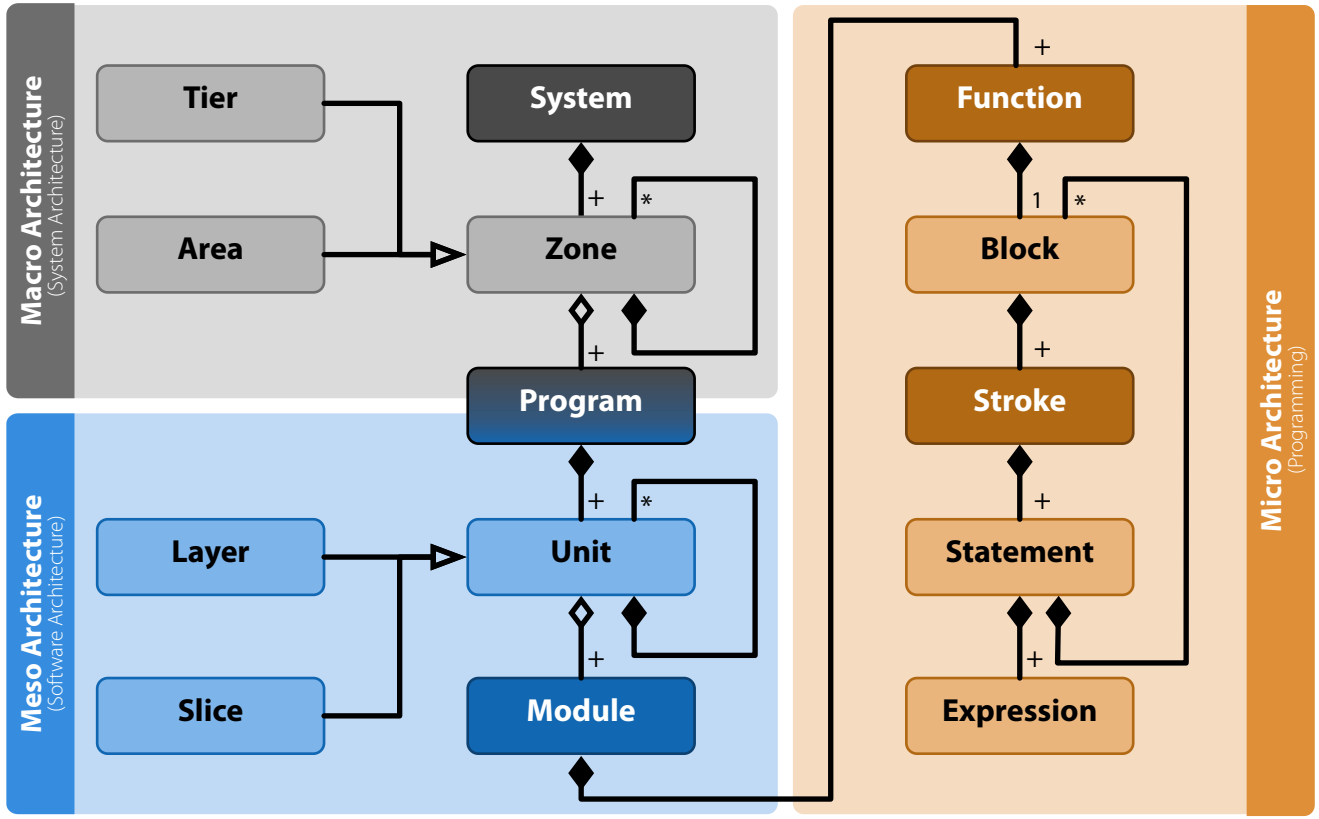

### HMR Human/Machine Responses

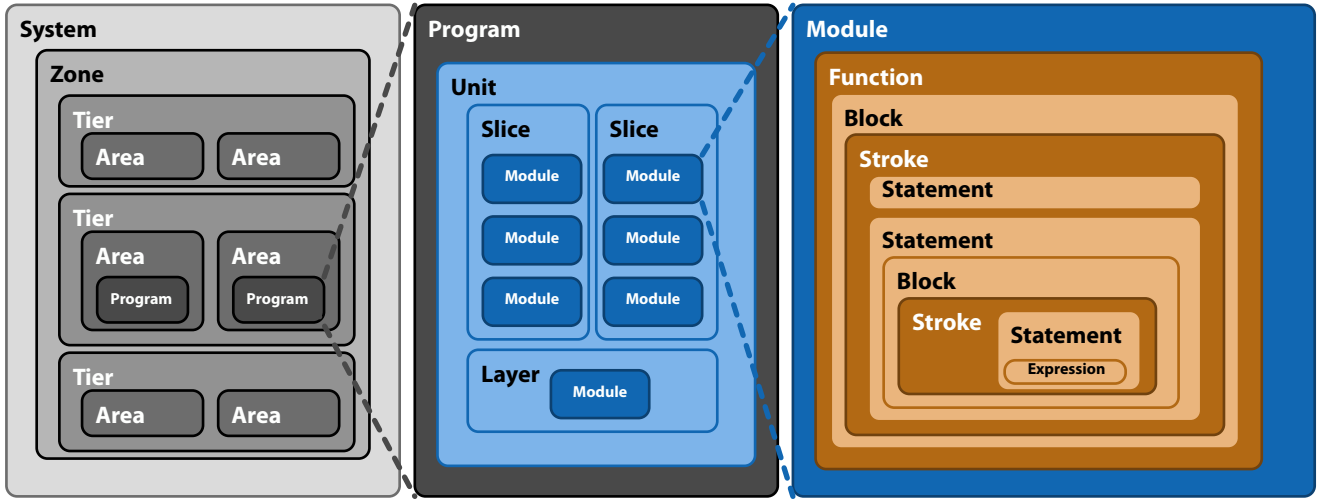
Support humans and machines in outputs through both description and parsing-free info.

ARCHITECTURE FUNDAMENTALS

# Component Hierarchy

TECHNISCHE UNIVERSITÄT MÜNCHEN




**Zone:**
Structuring **Tier** or **Area** within a **System** (or within an outer **Zone**).

**Unit:**
Structuring **Layer** or **Slice** within a **Program** (or within an outer **Unit**).

**Tier:**
Special kind of ranked **Zone** for horizontally grouping (logically or spatially distinct and network-linked) **Programs** or other **Tiers** and **Areas**.

**Layer:**
Special kind of ranked **Unit** for horizontally grouping (logically distinct and memory-linked) **Modules** or other **Slices** and **Layers**.

**Area:**
Special kind of unranked **Zone** for vertically grouping (logically or spatially distinct and network-linked) **Programs** or other **Tiers** and **Areas**.

**Slice:**
Special kind of unranked **Unit** for vertically grouping (logically distinct and memory-linked) **Modules** or other **Slices** and **Layers**.

**System:**
One or more network-linked **Programs**, forming a solution as a whole and the primary unit of thinking at the level of Macro Architecture.

**Module:**
Know-how encapsulating, potentially reusable group of **Functions** and primary unit of thinking at the level of Meso Architecture.

**Program:**
Stand-alone, self-contained, executable piece of software and primary unit of thinking at the level of both Macro and Meso Architecture.

**Function:**
Operation of a **Module**, with a well-defined interface, and one of the two primary units of thinking at the level of Micro Architecture.

**Statement:**
Execution of a single action, specified through one or more **Expressions** and zero or more inner **Blocks**.

**Stroke:**
Logical sequence of **Statements** in a **Block** and one of the two primary units of thinking at the level of Micro Architecture.

**Block:**
Sequence of one or more **Strokes** within a **Function** or an outer **Statement** and scope for variables of inner **Statements**.

**Expression:**
Simple or complex representation of a single data value.

ARCHITECTURE FUNDAMENTALS

# Architecture Patterns

TECHNISCHE UNIVERSITÄT MÜNCHEN

AF 06.1

## Pattern Definition

**Pattern:** unique, catchy, and associative **Name** and concise, precise, and normative **Description** of a contextual, regularly occurring, and practically relevant **Problem** and a general, highly reusable, and best practice **Solution** for it.

## Pattern Structure

Pattern

- **Name**
  - unique → has to be unambiguously distinguished
  - catchy → has to be recognized and memorized
  - associative → has to be intuitively associated to the solution
- **Description**
  - concise → has to be reasonably remembered
  - precise → has to be clear and unambiguous
  - normative → has to be foundational and stringent in expressiveness
- **Problem**
  - contextual → has to be described in a given context
  - regularly occurring → has to be motivated by enough situations
  - practically relevant → has to be motivated by enough practice
- **Solution**
  - general → has to be general enough to be sufficiently reusable
  - highly reusable → has to be applicable also in variants of the context
  - best practice → has to be considered a best or at least good practice

## Pattern Rationale

ARCHITECTURE FUNDAMENTALS

# Layer Architectures

(Patterns)

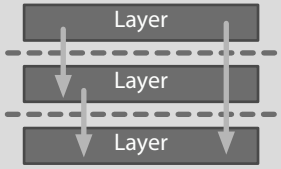
TECHNISCHE UNIVERSITÄT MÜNCHEN

AF 06.2

## Layering Principle

Horizontally split code or data into two or more logically, optionally also spatially, clearly distinct, isolating, named, and ranked Layers.

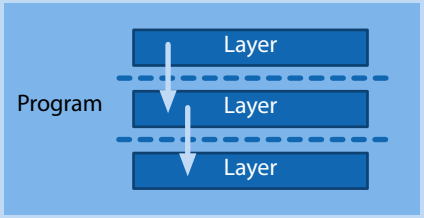
A Layer is not allowed to have relationships to or knowledge about any upper Layers. Additionally, for *Closed Layering,* each Layer is allowed to have relationships to and knowledge about the *directly* lower Layer only. In contrast to *Open Layering* or *Leaky Abstraction*, where each Layer is allowed to have relationships to and knowledge about *any* lower Layer.




## LR Layer

Split related code or data of a Program into two or more logically distinct domain- or technology-induced Layers.

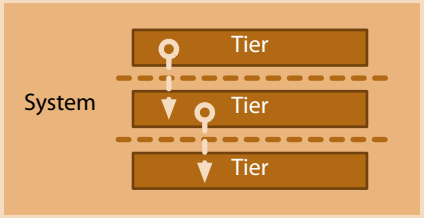
Rationale: Separation of Concern, Single Responsibility Principle, Mastering Complexity, Change Isolation, Functional Abstraction.




## TR Tier

Split related code or data of a System into two, three or more logically and spatially distinct, network-connected, domain- or technology-induced Tiers.

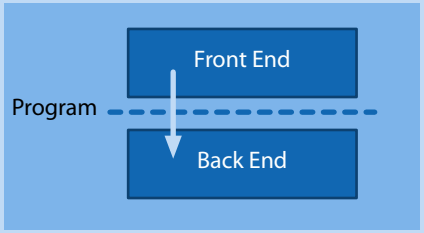
Rationale: Separation of Concern, Single Responsibility Principle, Mastering Complexity, Change Isolation, Functional Abstraction, Deployment Partitioning.




## FB Front End / Back End

Split the code of a Program into exactly two logical Layers: a user-facing Front End and a data-facing Back End.

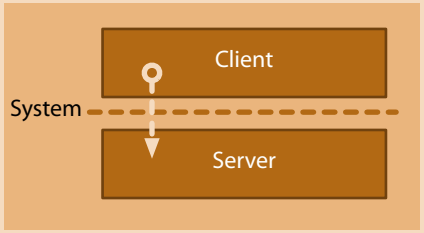
Rationale: Separation of Concern, Single Responsibility Principle, Mastering Complexity, Change Isolation, Functional Abstraction, Organisational Alignment.




## CS Client / Server

Split the code of a System into two spatially distinct, network-connected Layers, each forming a stand-alone Program: a user-facing and multi-instantiated (Rich) Client and a data-facing (and logically) single-instantiated (Thin) Server. Both contain a Front/Back End.

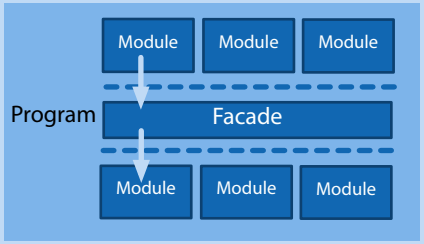
Rationale: Multi-User, User Computing Resource Leverage, Distributed Computing.




## FD Facade

Splice a domain-specific Facade Layer into two Layers of two or more Modules. The extra Facade Layer acts as a broker between the Modules.

Rationale: Information Hiding, Cross-Cutting Concern Centralization, Functionality Orchestration, Authorization, Validation, Conversion.




## MW Middleware

Splice a domain-unspecific Middleware Layer into a Client/Server communication. The extra Layer is a stand-alone Program Tier and acts as a broker between Client and Server.

Rationale: Communication Peer Discovery Simplification, Transport Protocol Conversions, Network Topology Flexibility.