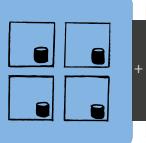
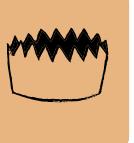
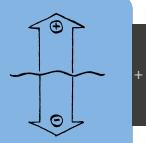
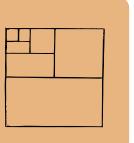
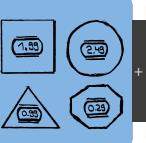
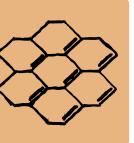
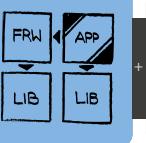
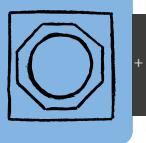
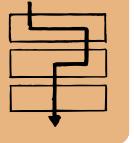
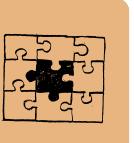
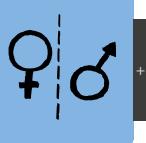
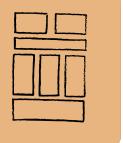
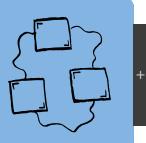
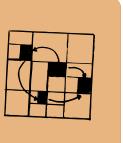
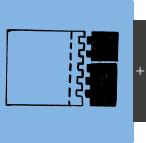
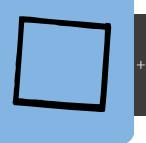
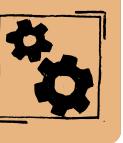
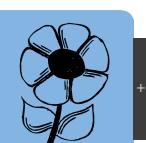




# Software Engineering in der industriellen Praxis (SEIP)

Dr. Ralf S. Engelschall

# Architecture Principles

<b>FL</b> Factual Locality	Resources are as spatially and temporarily local-scoped to solution components as possible		<b>ES</b> Exclusive Sovereignty	Exclusive resource sovereignty by the enclosing component	
<b>CA</b> Contextual Adequacy	Neither insufficient nor exaggerated solutions for each context		<b>SP</b> Solution-oriented Proportionality	Good expected proportionality in each solution context	
<b>HC</b> Holistic Consistency	Full consistency across all aspects of a solution		<b>SH</b> Structural Homogeneity	Maximum homogeneity in the structure of a solution	
<b>CR</b> Constructional Reusability	High reuse of proven structural components and partial solutions		<b>FS</b> Fulfilled Standards	Compliance to standards as much as possible, as long as the benefits predominate the drawbacks	
<b>FA</b> Functional Abstraction	Suitable level of abstraction across all functional aspects of a solution		<b>FT</b> Functional Traceability	Suitable traceability across all functional aspects of a solution	
<b>CI</b> Communicative Interoperability	Maximum interoperability in communication between solutions		<b>EH</b> Environmental Harmony	Maximum harmony in the integration of the solution with its environment	
<b>AR</b> Avoided Redundancy	Minimum total number of copies of a single resource		<b>MS</b> Minimum Special-Cases	Minimum total number of special-cases in a solution	
<b>LS</b> Logical Separation	Separation of concerns between the components of a solution		<b>SM</b> Structural Modularity	Splitting of a solution into manageable structural components	
<b>LC</b> Loose Coupling	Loose coupling in communication and referencing between solution components		<b>SC</b> Strong Cohesion	Strong relationship between functionalities within a single solution component	
<b>OE</b> Open Extensibility	Solution components can be extended by third-parties at fixed interfaces		<b>CC</b> Closed Changeability	Solution components are protected against direct change by third-parties	
<b>UI</b> Unique Identification	Unique identification of all components of a solution		<b>UA</b> Uniform Addressing	Uniform addressing of all components of a solution	
<b>OS</b> Overall Simplicity	All design aspects of a solution are as simple as possible and only as complicated as necessary		<b>EC</b> Encapsulated Complexity	Complex related aspects of a solution are encapsulated into a single responsible component	
<b>LA</b> Least Astonishment	All design aspects of a solution are as little astonishing as possible and only as esoteric as necessary		<b>SD</b> Self Documentation	All design aspects of a solution are preferably self-documenting	
<b>OD</b> Operational Delight	The solution provides users true delight even on long-term operation		<b>AA</b> Artistic Aesthetics	The solution has holistic aesthetics and artistic love in details	

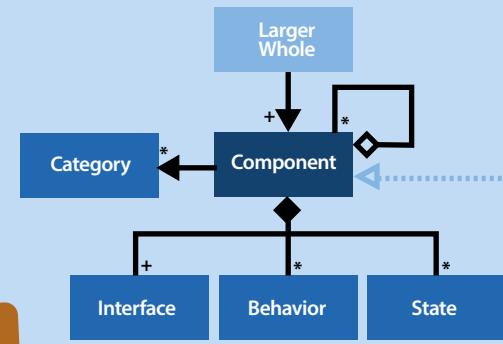
## Definition of a Component (of a Larger Whole):

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.

### Example Categories of Components:

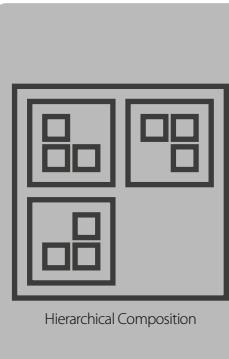
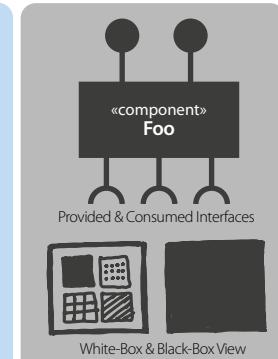
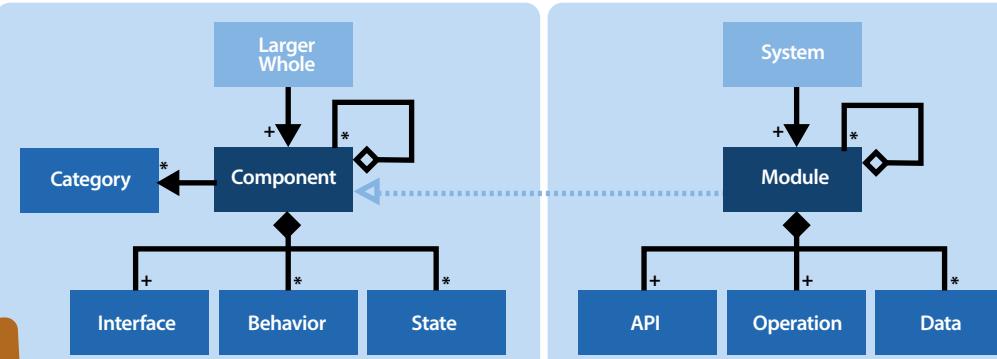
- Namespace
- Directory, File
- Configuration, Section, Directive
- Host, Virtual Machine, Container
- Process Group, Process, Thread
- Application, Microservice, Program
- Package, Class, Function
- Database, Schema, Table, Record
- Datamodel, Entity Group, Entity
- User Interface, Dialog, Widget

*Any group of anything!*



## 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.



## How to find Components (or Modules)?

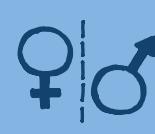
### DCA Domain Concept Abstraction

Model domain concepts as entity components and then group at higher levels.



### SOC Separation of Concerns

Build components for clearly distinct concerns.



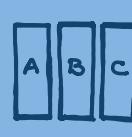
### USE Reusability Potential

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.



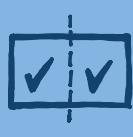
### SRP Single Responsibility Principle

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.



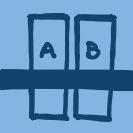
### CNC Coupling and Cohesion

Decide on components based on their loose coupling and strong cohesion.



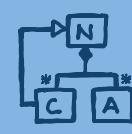
### CCC Cross-Cutting Concerns

Build common cross-cutting concerns as cross-cutting components.



### OOD Object-Oriented Design

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



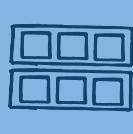
### DEP Dependency Encapsulation

Decide on components based on their encapsulation of dependencies.



### PAT Architecture Patterns

Build inner components to comply to outer structure, slicing and clustering architectures.



# Interface Design

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

Institutional Content: Version 1.0.9 (2022-1-15) Authorised 2015-2022 by Dr. Ralf S. Engelschall <<http://engelschall.com>>. All Rights Reserved.  
Unauthorized Reproduction Prohibited. Licensed to Technische Universität München (TUM) for reproduction in Computer Science lecture contexts only.

## Types of Software Interfaces

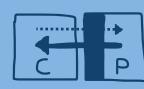
### API Application Programming Interface

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`



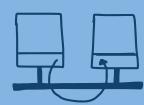
### GUI Graphical User Interface

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.



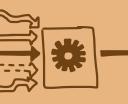
### OC Orthogonal & Concise

Supports combinatorial use-cases, causes minimum boilerplate.



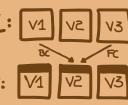
### TP Tolerant & Predictable

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.



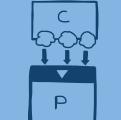
### EVE Event Emitter

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.

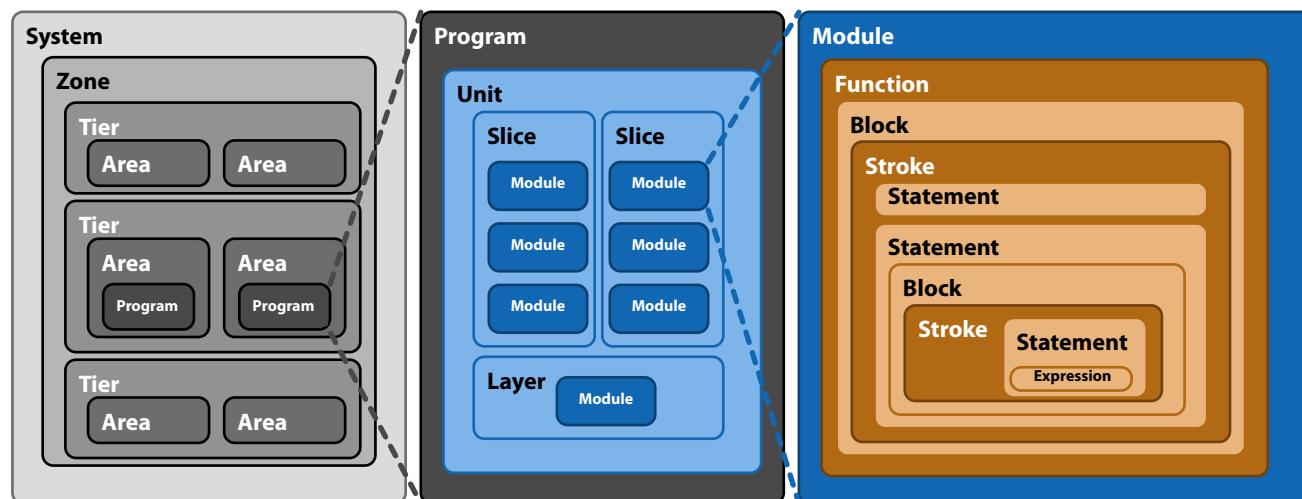
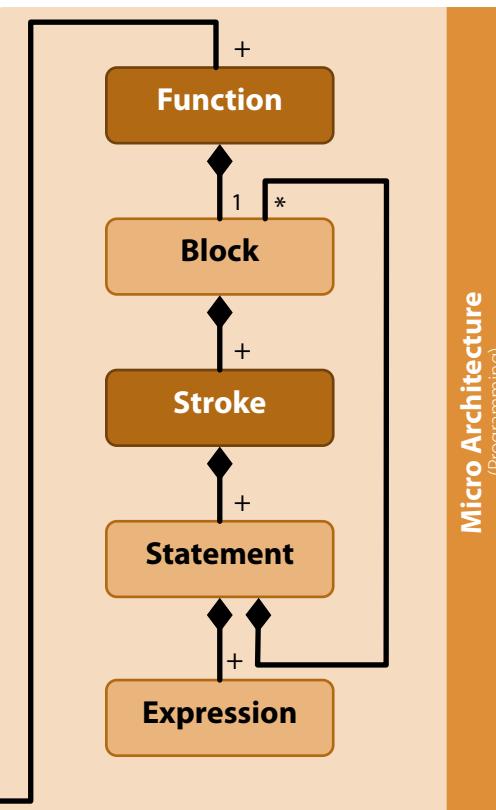
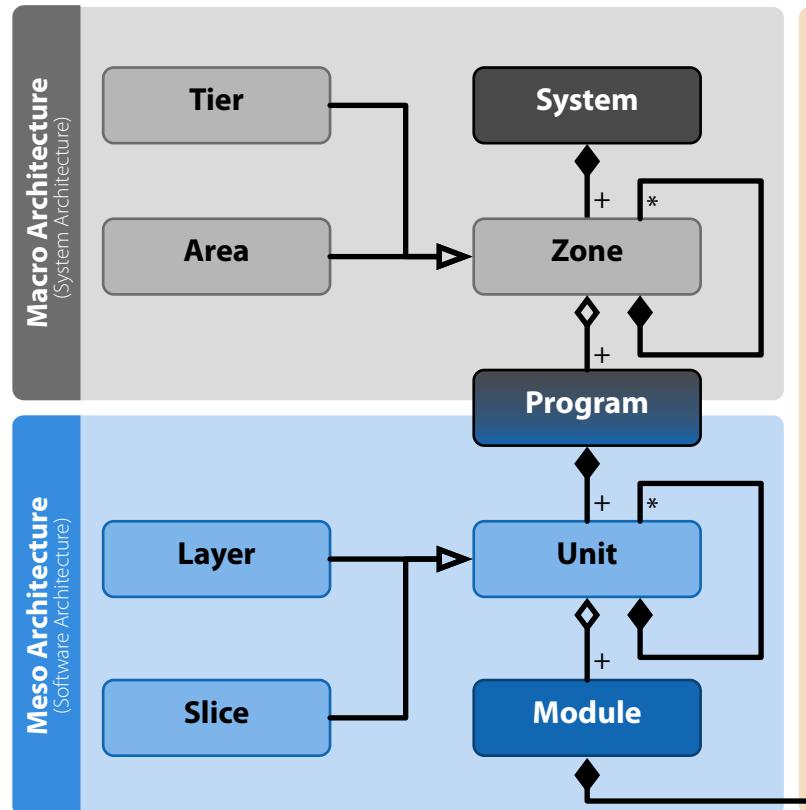


### HMR Human/Machine Responses

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



# Component Hierarchy



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

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

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

**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.

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

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

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

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

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

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

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

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

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

## 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

catchy

associative

Description

concise

precise

normative

Problem

contextual

regularly occurring

practically relevant

Solution

general

highly reusable

best practice

### Pattern Rationale

→ has to be unambiguously distinguished

→ has to be recognized and memorized

→ has to be intuitively associated to the solution

→ has to be reasonably remembered

→ has to be clear and unambiguous

→ has to be foundational and stringent in expressiveness

→ has to be described in a given context

→ has to be motivated by enough situations

→ has to be motivated by enough practice

→ has to be general enough to be sufficiently reusable

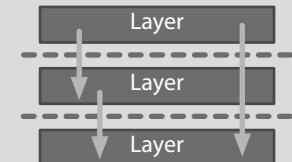
→ has to be applicable also in variants of the context

→ has to be considered a best or at least good practice

## Layering Principle

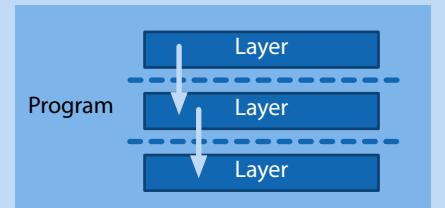
Horizontally split code or data into two or more logically, optionally also spatially, clearly distinct, isolating, named, and ranked Layers.

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

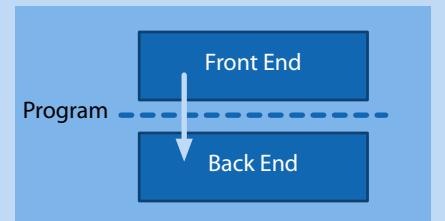
Split related code or data of a Program into two or more logically distinct domain- or technology-induced Layers.



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

## FB Front End / Back End

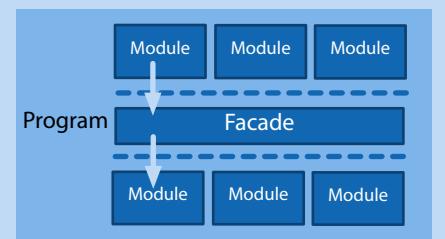
Split the code of a Program into exactly two logical Layers: a user-facing Front End and a data-facing Back End.



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

## 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.

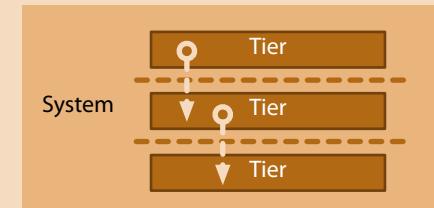


Rationale: Information Hiding, Cross-Cutting Concern Centralization, Functionality Orchestration, Authorization, Validation, Conversion.

## 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.

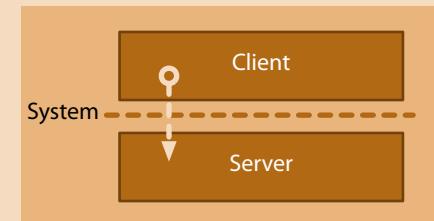
Rationale: Separation of Concern, Single Responsibility Principle, Mastering Complexity, Change Isolation, Functional Abstraction, Deployment Partitioning.



## 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.



## 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.

