































Software Engineering in der industriellen Praxis (SEIP)

Dr. Ralf S. Engelschall

FL Factual Locality Resources are as spatially and temporally local-scoped to solution components as possible 	ES Exclusive Sovereignty Exclusive resource sovereignty by the enclosing component 	LS Logical Separation Separation of concerns between the components of a solution 	SM Structural Modularity Splitting of a solution into manageable structural components 
CA Contextual Adequacy Neither insufficient nor exaggerated solutions for each context 	SP Solution-oriented Proportionality Good expected proportionality in each solution context 	LC Loose Coupling Loose coupling in communication and referencing between solution components 	SC Strong Cohesion Strong relationship between functionalities within a single solution component 
HC Holistic Consistency Full consistency across all aspects of a solution 	SH Structural Homogeneity Maximum homogeneity in the structure of a solution 	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 
CR Constructural Reusability High reuse of proven structural components and partial solutions 	FS Fulfilled Standards Compliance to standards as much as possible, as long as the benefits predominate the drawbacks 	UI Unique Identification Unique identification of all components of a solution 	UA Uniform Addressing Uniform addressing of all components of a solution 
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 	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 
CI Communicative Interoperability Maximum interoperability in communication between solutions 	EH Environmental Harmony Maximum harmony in the integration of the solution with its environment 	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 
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 	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 

In der IT Architecture folgt man **Architecture Principles**, welche grundlegende Prinzipien und Vorgehen zusammenfassen. Man kennt 28 Prinzipien, in 14 Pärchen gruppiert werden können, da immer zwei Prinzipien inhaltlich eine starke Nähe aufweisen. Der Architekt soll den Prinzipien generell folgen, darf sie aber verletzen, solange er einen guten Grund dafür hat. Der beste Grund wäre eine spezielle projektspezifische Anforderung.

Beachte: das Prinzip **Logical Separation** (aka **Separation of Concern**) ist eines der wichtigsten, da aus ihm etliche andere Prinzipien fast automatisch folgen, unter anderem z.B. **Structural Modularity**.

Hinweis: die Prinzipien **Loose Coupling** und **Strong Cohesion** sind in der Literatur als das Kombi-Prinzip "Loose Coupling, Strong Cohesion" bekannt. Die Prinzipien **Open Extensibility** und **Closed Changeability** sind in der Literatur als das Kombi-Prinzip "Open-Close" bekannt.

Beachte: das Prinzip **Overall Simplicity** ist eines der schwersten umzusetzenden Prinzipien, da nichts in der IT wirklich einfach ist. Alles erscheint nur solange einfach, solange man noch nicht genügend davon versteht. Danach muss man es erst wieder mühsam neu "einfach" machen. Das ist die Kunst bei Architektur: schwierige Dinge zu vereinfachen! Wenn etwas nicht viel weiter vereinfacht werden kann und immer noch gewisse Komplexität aufweist, kann man es zumindest mit dem Prinzip **Encapsulated Complexity** versuchen zu verschatten.

Fragen

? Zählen Sie mindestens 4 wesentliche **Architecture Principles** auf!

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.

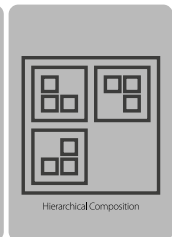
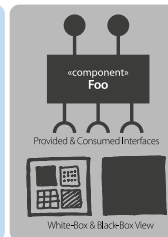
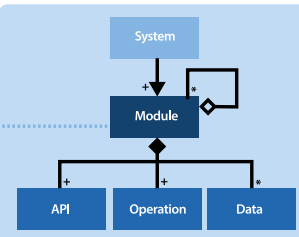
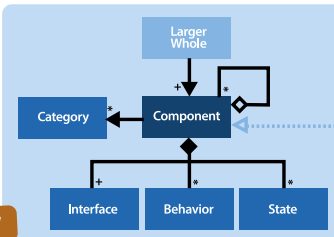
Definition of a **Module** (of a System):

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.	

In der Software Architecture dreht sich alles um Komponenten (**Components**) und Schnittstellen (**Interfaces**). Deshalb ist **Component Design** eine zentrale Aufgabe des Architekten.

Eine Component **kapselt** ein bestimmtes **Know-How**, ist **potenziell wiederverwendbar** und **ersetzbar**. Eine Component besitzt ein **Verhalten** und einen **Zustand** und verbirgt die interne Komplexität von beidem hinter "kleinen" **vertraglichen Schnittstellen**. Sie stellt ihren Mehrwert über die Differenz bereitgestellter und konsumierter Schnittstellen bereit. Sie kann als **Whitebox** oder **Blackbox** betrachtet werden, abhängig davon, ob man ihre internen Details von außen sieht oder nicht. Components sind hierarchisch angeordnet, gehören eventuell zu bestimmten **Kategorien** und besitzen **explizite Abhängigkeiten** untereinander.

Man unterscheidet zwischen dem allgemeineren Konzept der **Component** ("any group of anything") und dem spezielleren Konzept des (über Source Code definierten) **Module**.

Components findet man über zahlreiche Wege. Die meisten leiten sich direkt aus bestehenden Methoden, Prinzipien oder Mustern ab. Die beiden wichtigsten Wege für einen Komponentenschnitt in der Praxis sind: **Separation of Concerns** (welches einmalige Anliegen bzw. welche einmalige Aufgabe hat die Komponente?) und **Single Responsibility Principle** (welche einmalige Verantwortung hat die Komponente?).

Fragen

- ❓ Zählen Sie mindestens 7 Eigenschaften/Aspekte auf, die eine Komponente besitzt!
- ❓ Was sind die beiden wichtigsten Wege, um in der Praxis einen Komponentenschnitt zu finden?

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, ...	
Types of Software Interfaces		Characteristics of Good Interfaces		Selected Interface Design Patterns	
API Application Programming Interface Example: foo("bar", 42) (call and use)				IVF Interface Version & Features V1.2 Provide version and feature information for algebraic comparison and feature detection.	
SPI Service Provider Interface Example: register("foo", (x,k) => ...) (extend and implement)		AP Appropriate & Proportional Appropriate to consumer requirements, proportional to provider functionality.			
SCI Startup Configuration Interface Examples: INI, Java Properties, TOML, YAML, JSON, XML, etc.		SA Shielding & Abstracting Shields from direct access, abstracts and hides implementation details.		2LF Leaky Two-Layer Facade Provide higher-level convenient use-case and lower-level orthogonal feature interface.	
		IE Inviting & Expressive Invites through "outside-in" design, powerful in expressiveness.		EVE Event Emitter Emit events to previously registered, interested consumers.	
BPI Batch Processing Interface Examples: Unix at(1), Unix ts(1), GNU Batch, Spring Batch, Java Batch, SAP LO-BM, etc.		IF Intuitive & Foolproof Intuitive to grasp and use, hard to misuse.		CTX Multi-Context Use contexts to distinguish between different usage scenarios and to carry common info.	
		OC Orthogonal & Concise Supports combinatorial use-cases, causes minimum boilerplate.		CEF Configure-Execute Flow Spread use-cases onto a flow of configuration exchanges and a final executional exchange.	
CLI Command-Line Interface Example: foo -x --bar=baz quux				IOC Inversion Of Control Invert control on asynchronous operations via callbacks, promises or async. mechanisms.	
GUI Graphical User Interface Examples: Windows/WPF, macOS/Cocoa, KDE/Qt, GNOME/GTK		TP Tolerant & Predictable Tolerant on input, predictable on output.			
RNI Remote Network Interface Examples: GraphQL/HTTP, REST, SOAP, RMI, WebSockets, AMQP, MQTT, etc.		EC Extensible & Compatible Easy to extend for providers, backward/forward-compatible for consumers.		HMR Human/Machine Responses 302 MOVED TEMPORARILY Support humans and machines in outputs through both description and parsing-free info.	

Eine Schnittstelle (**interface**) ist eine wohldefinierte (**well-defined**), abschirmende (**shielding**), abstrahierende (**abstracting**) Berandung (**boundary**) einer passiven bereitstellenden Komponente (**component**), welche aus einer oder mehreren klarunterscheidbaren Interaktionsendpunkten (**interaction endpoints**) besteht.

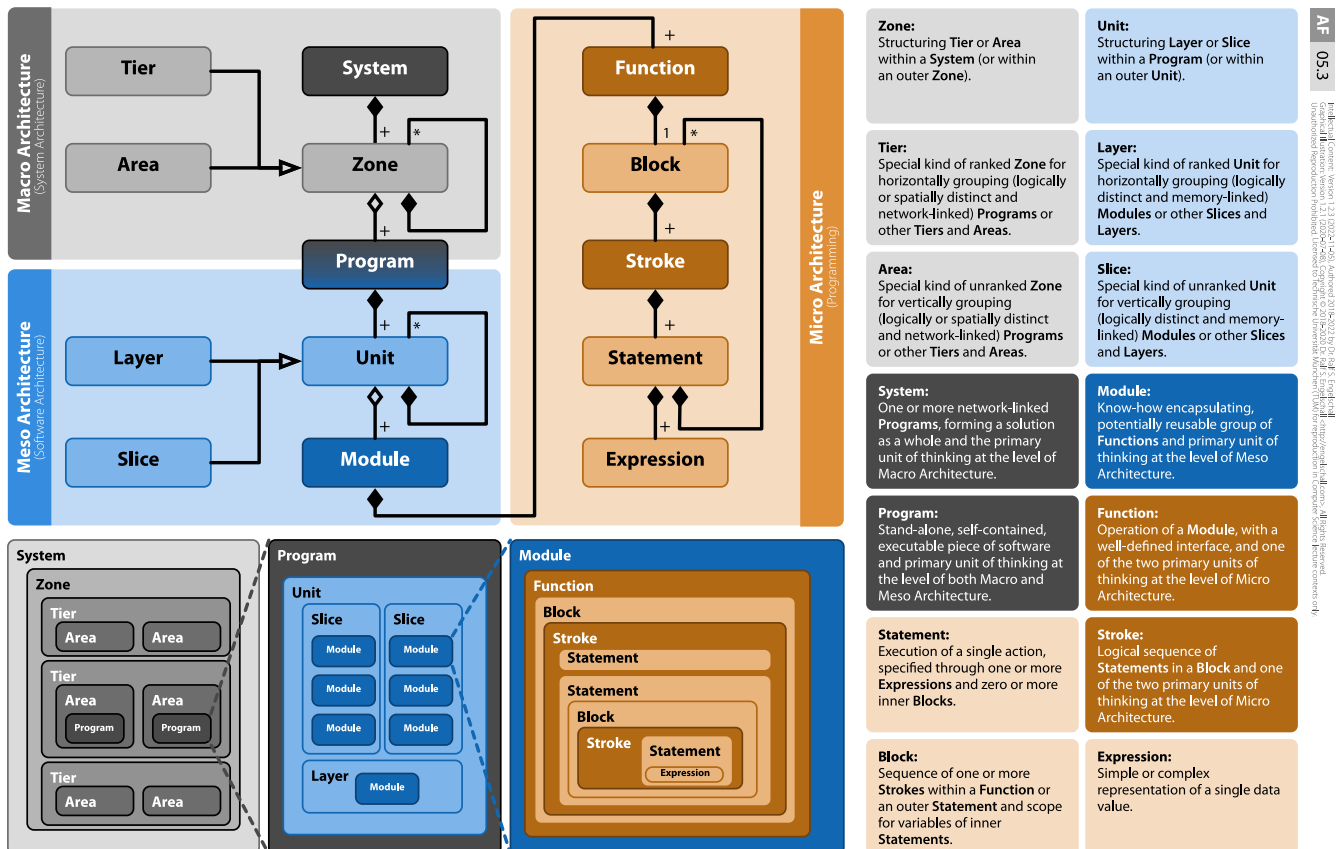
Auf jeden Interaktionsendpunkt wird dabei von einer aktiven konsumierenden Komponente durch den Austausch (**exchange**) von **input/output** Informationen (**information**) zugegriffen und wird unter einem bestimmten syntaktischen (**syntactical**) und semantischen (**semantical**) Vertrag (**contract**) betrieben.

Es gibt zahlreiche Arten von Schnittstellen, die alle dieser Definition entsprechen. "Gute" Schnittstellen haben darüber hinaus bestimmte Eigenschaften/ Charakteristiken. Die vier der besten Eigenschaften sind: **Proportional** (die Schnittstelle ist kleiner und in der Größe proportional zur dahinterliegenden Funktionalität), **Expressive** (die Schnittstelle stellt ein leistungsfähiges Programmiermodell zur Verfügung), **Orthogonal** (die Schnittstelle erlaubt kombinatorische Use-Cases), und **Concise** (die Schnittstelle erzeugt wenig "Boilerplate Code" bei der Nutzung).

Es gibt zahlreiche Software Pattern für Schnittstellen. Ein interessantes Muster ist die **Leaky Two-Layer Facade**, bei dem eine Bibliothek zwei Schnittstellen besitzt: eine obere, bequeme und Use-Case-bezogene Schnittstelle und eine untere, orthogonale Feature-bezogene Schnittstelle. Der Witz ist, daß die obere durch alleine die untere Schnittstelle implementiert wird und daß die untere Schnittstelle "durchscheint" ("leaky" bzw. Open Layering).

Fragen

- 🔍 Zählen Sie mindestens 8 Eigenschaften/Aspekte auf, die eine Schnittstelle definieren!
- 🔍 Zählen Sie mindestens 4 Eigenschaften/ Charakteristiken von guten Schnittstellen auf!



Eine **Component** ist "any group of anything" in der Software Architecture. Dennoch gibt es prominente Komponenten-kategorien, die eine bestimmte allgegenwärtige **Component Hierarchy** im Software Engineering bilden. Diese besteht aus den drei Ebenen **Macro Architecture** (aka System Architecture), **Meso Architecture** (aka Software Architecture) und **Micro Architecture** (aka Programming).

In der Ebene Macro Architecture hat man mit **Systems** (aka Applications) zu tun, die aus hierarchisch angeordneten, infrastrukturellen **Zones** bestehen, welche entweder (horizontale) **Tiers** oder (vertikale) **Areas** sein können. Die **Zones** bestehen ihrerseits aus **Programs**.

Diese **Programs** bestehen in der Ebene Meso Architecture wiederum aus hierarchisch angeordneten **Units**, welche entweder (horizontale) **Layer** oder (vertikale) **Slices** sein können. Die **Units** bestehen ihrerseits aus **Modules**.

Die **Modules** bestehen in der Ebene Micro Architecture wiederum aus **Functions** und diese aus hierarchisch angeordneten (lexikalischen) **Blocks**, welche ihrerseits aus **Strokes** (aka "Thoughts") bestehen, welche ihrerseits aus **Statements** bestehen und diese bestehen am Ende aus einzelnen **Expressions**.

Die vier **Primary Units of Thinking** sind **Systems**, **Programs**, **Modules** und **Strokes**.

Fragen

- ❓ Welche drei Komponentenkategorien kennt man auf der Ebene der Macro Architecture (aka System Architecture)?
- ❓ Welche drei Komponentenkategorien kennt man auf der Ebene der Meso Architecture (aka Software Architecture)?
- ❓ Welche fünf Komponentenkategorien kennt man auf der Ebene der Micro Architecture (aka Programming)?

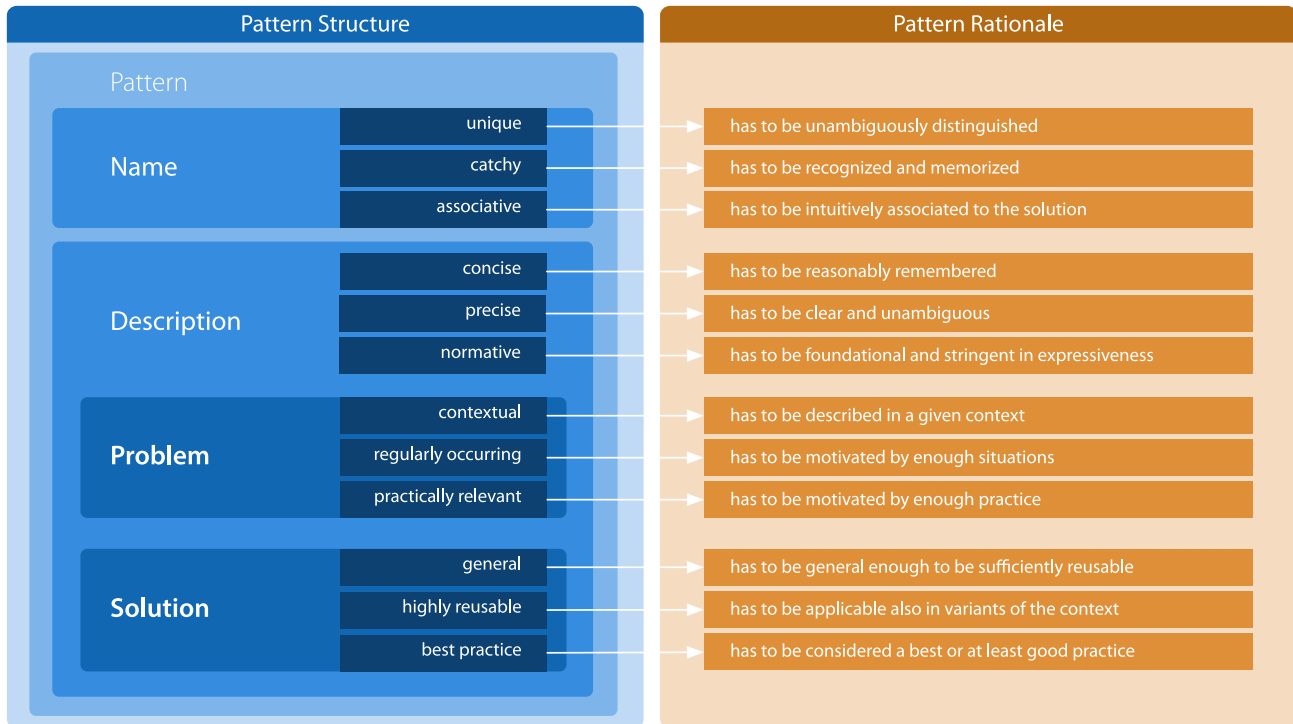
Pattern Definition

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.



AF 06.1

Indicate the version of the document (AF 06.1) and the version of the software (AF 06.1) used for the generation of the document. The document is licensed under the Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License. For more information, please visit <https://creativecommons.org/licenses/by-nc-sa/4.0/>.



Definition eines **Architekturmusters**: einzigartiger, eingängiger und assoziativer Name und prägnante, präzise und normative Beschreibung eines kontextuellen, regelmäßig auftretenden und praktisch relevanten Problems und einer allgemeinen, hochgradig wiederverwendbaren und praxisbewährten Lösung für dieses Problem.

Die Begründungen ist, daß ein **Architekturmuster**: eindeutig unterschieden werden muß, wiedererkannt und erinnert werden muß, intuitiv mit der Lösung assoziiert werden muß, sich gut merken lassen muß, klar und eindeutig sein muß, grundlegend und stringent in der Ausdrucksfähigkeit sein muß, in einem bestimmten Kontext beschrieben werden muß, durch genügend Situationen motiviert sein muß, durch genügend Praxis motiviert sein muß, um ausreichend wiederverwendbar zu sein auch allgemein genug sein muß, auch in Varianten des Kontexts anwendbar sein muß, und als beste oder zumindest gute Praxis angesehen werden muß.

Architekturmuster ermöglichen es einem insbesondere effizient zu kommunizieren (Name) und von den eingeflossenen Erfahrungen zu profitieren (Best Practice).

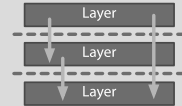
Fragen

? Warum sind **Architekturmuster** interessant?

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.



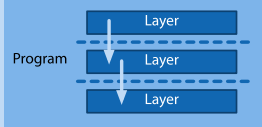
AF 062

Copyright © 2020, TUM School of Management. All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or by any information storage and retrieval system, without prior written permission from the TUM School of Management.

LR Layer

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

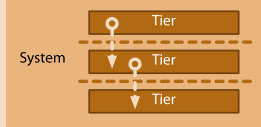
Rationale: Separation of Concern, Single Responsibility Principle, Mastering Complexity, Change Isolation, Functional Abstraction.



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.

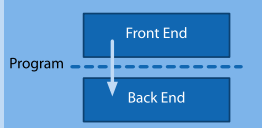
Rationale: Separation of Concern, Single Responsibility Principle, Mastering Complexity, Change Isolation, Functional Abstraction, Deployment Partitioning.



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.

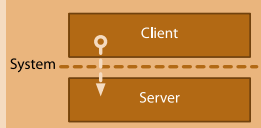
Rationale: Separation of Concern, Single Responsibility Principle, Mastering Complexity, Change Isolation, Functional Abstraction, Organisational Alignment.



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.

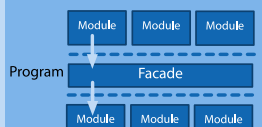
Rationale: Multi-User, User Computing Resource Leverage, Distributed Computing.



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.

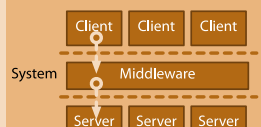
Rationale: Information Hiding, Cross-Cutting Concern Centralization, Functionality Orchestration, Authorization, Validation, Conversion.



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.



Beim **Layering** werden Code oder Daten in zwei oder mehr logische (**logically**) — ggf. auch “physikalische” (**spatially**) — **Layer** geschnitten. Diese Layer sind klar unterscheidbar (**clearly distinct**), von einander isoliert (**isolated**), benannt (**named**) und stehen in einer Rangordnung (**ranked**). Layer werden immer **horizontal** gezeichnet.

Ein Layer hat keine Beziehung (**relationship**) zu, oder Wissen (**knowledge**) über, irgendwelche Layer über ihm. Zusätzlich darf er, beim **Closed Layering**, eine Beziehung zu oder Wissen über den direkten Layer unter ihm haben. Außerdem darf er beim **Open Layering** bzw. der **Leaky Abstraction** sogar eine Beziehung zu oder Wissen über irgendwelche Layer unter ihm haben.

Falls sich das Layering über Netzwerk-Grenzen erstreckt oder eine “physikalische” Grenze durchbricht, spricht man nicht mehr von einzelnen Layern, sondern von **Tiers**.

Zerfällt ein Programm in einen vorderen bzw. auf das User-Interface fokussierenden Layer und einen hinteren bzw. auf Daten fokussierenden Layer, so spricht man bei den beiden Layern auch von **Front End** und **Back End** des Programms. Dies ist nicht zu verwechseln mit **Client** und **Server**, welche die zwei Tiers eines Systems über ihre speziellen Rollen benennt. Sowohl der Client als auch der Server sind eigenständige Programme mit jeweils einem Front End und einem Back End.

Eine sehr spezieller und prominenter Layer ist die **Facade**, welche innerhalb eines Programms die Module von zwei anderen Layern von einander trennt. Eine Variante der Facade auf Ebene eines Systems (statt auf der Ebene eines Programms) ist die **Middleware**, welche eine Client/Server-Kommunikation auftrennt.

Fragen

- ❓ Wie nennt man die entstehenden Einheiten, wenn Code oder Daten **horizontal** geschnitten werden?
- ❓ Was ist der Unterschied zwischen den Layer-Pärchen **Front/Back End** und **Client/Server**?