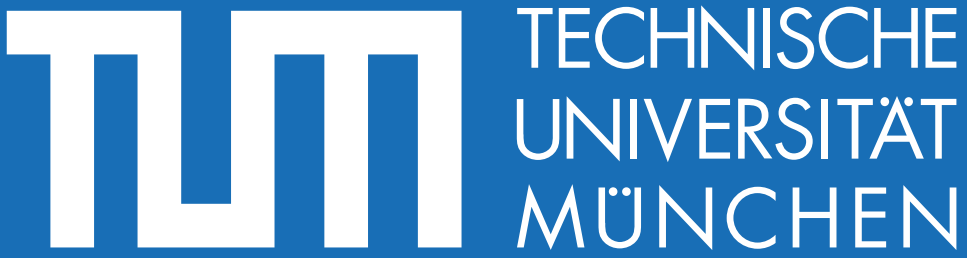
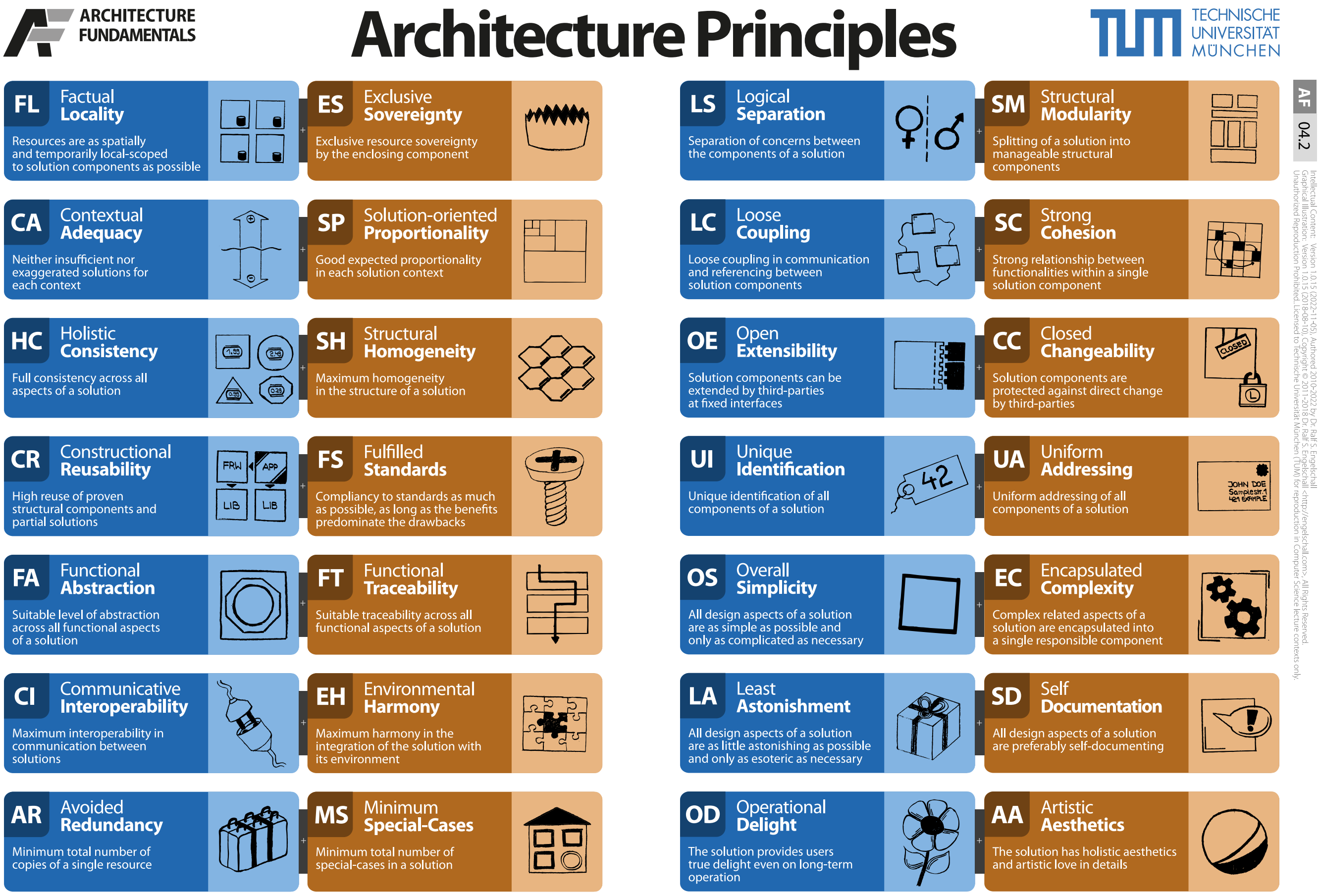


Software Engineering in der industriellen Praxis (SEIP)

Dr. Ralf S. Engelschall



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**.

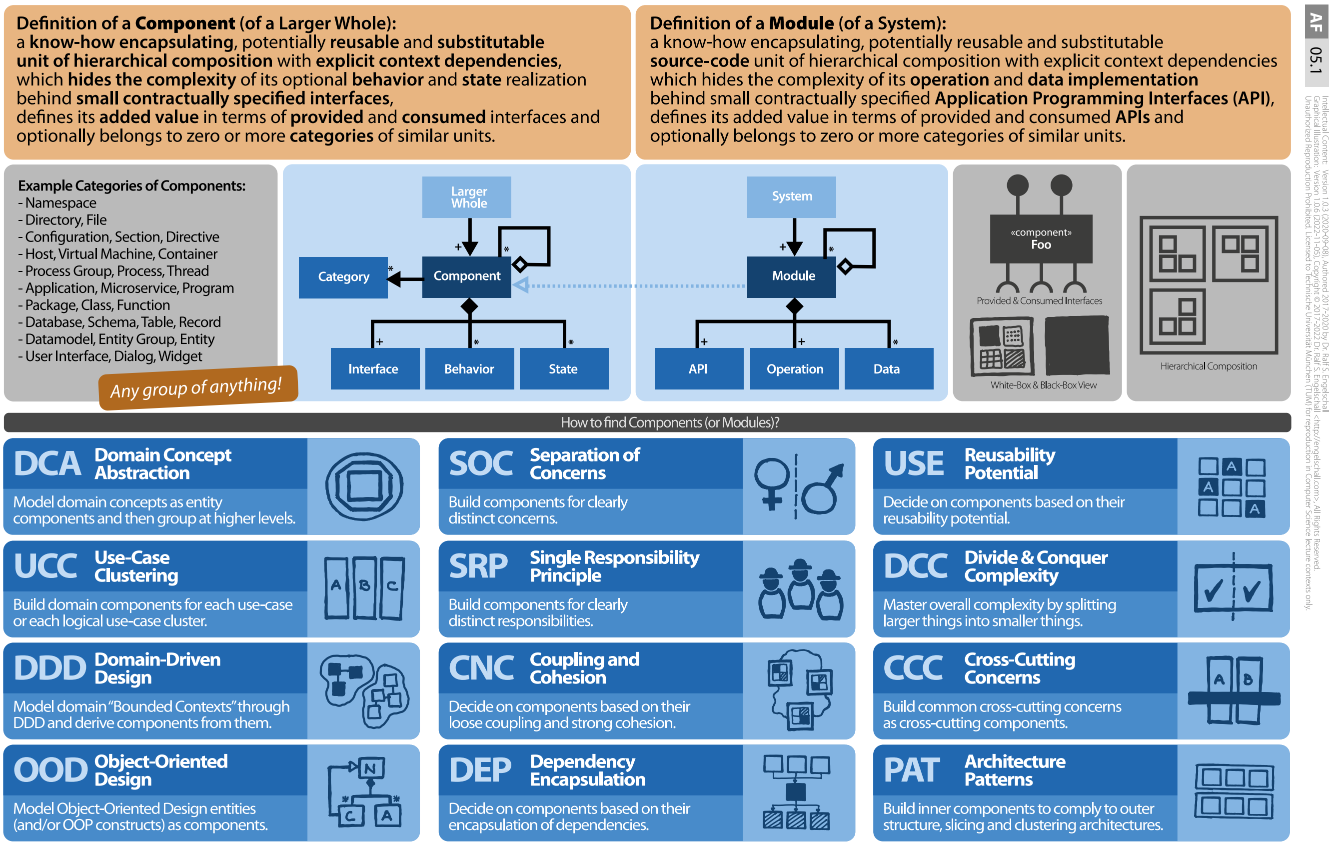
Hinweis: die Prinzipien **Loose Coupling** und **Strong Cohesion** sind in der Literatur als das Kombi-Prinzip “Loose Coupling, Strong Cohesion” bekannt. Die Prinzipien **Open Extensibility** und **Closed Changeability** sind in der Literatur als das Kombi-Prinzip “Open-Close” bekannt.

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!

Component Design



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

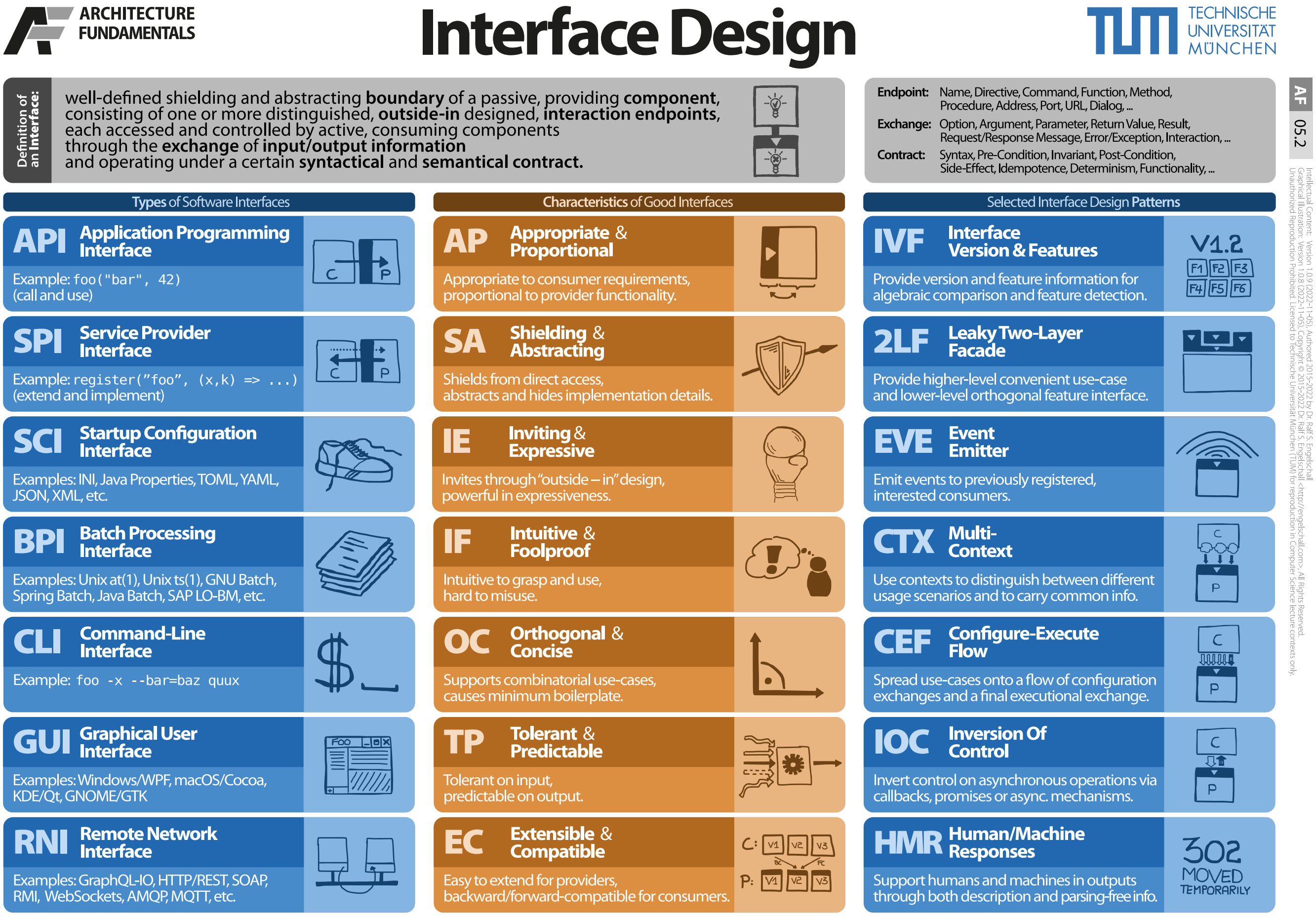
Eine Component **kapselt** ein bestimmtes **Know-How**, ist **potenziell wiederverwendbar** und **ersetzbar**. Eine Component besitzt ein **Verhalten** und einen **Zustand** und verbirgt die interne Komplexität von beidem hinter "kleinen" **vertraglichen Schnittstellen**. Sie stellt ihren Mehrwert über 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?



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.

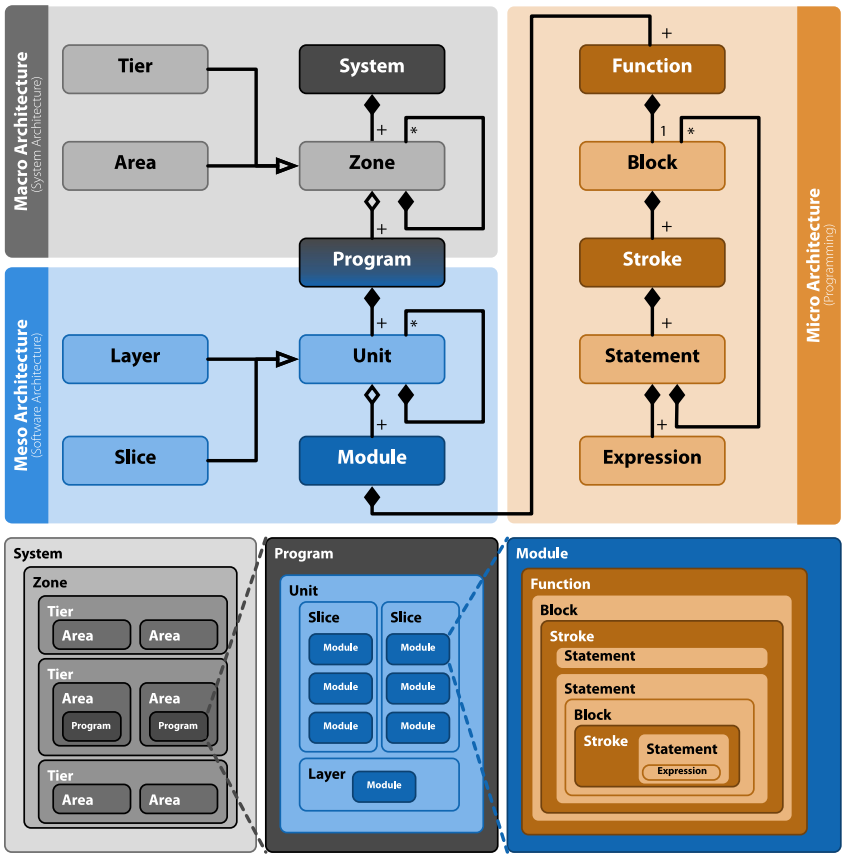
Es gibt zahlreiche Arten von Schnittstellen, die alle dieser Definition entsprechen. "Gute" Schnittstellen haben darüber hinaus bestimmte Eigenschaften/Charakteristiken. Die vier der besten Eigenschaften sind: **Proportional** (die Schnittstelle ist kleiner und in der Größe proportional zur dahinterliegenden 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!

Component Hierarchy



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.

Eine **Component** ist "any group of anything" in der Software Architecture. Dennoch gibt es prominente Komponentenkategorien, 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**.

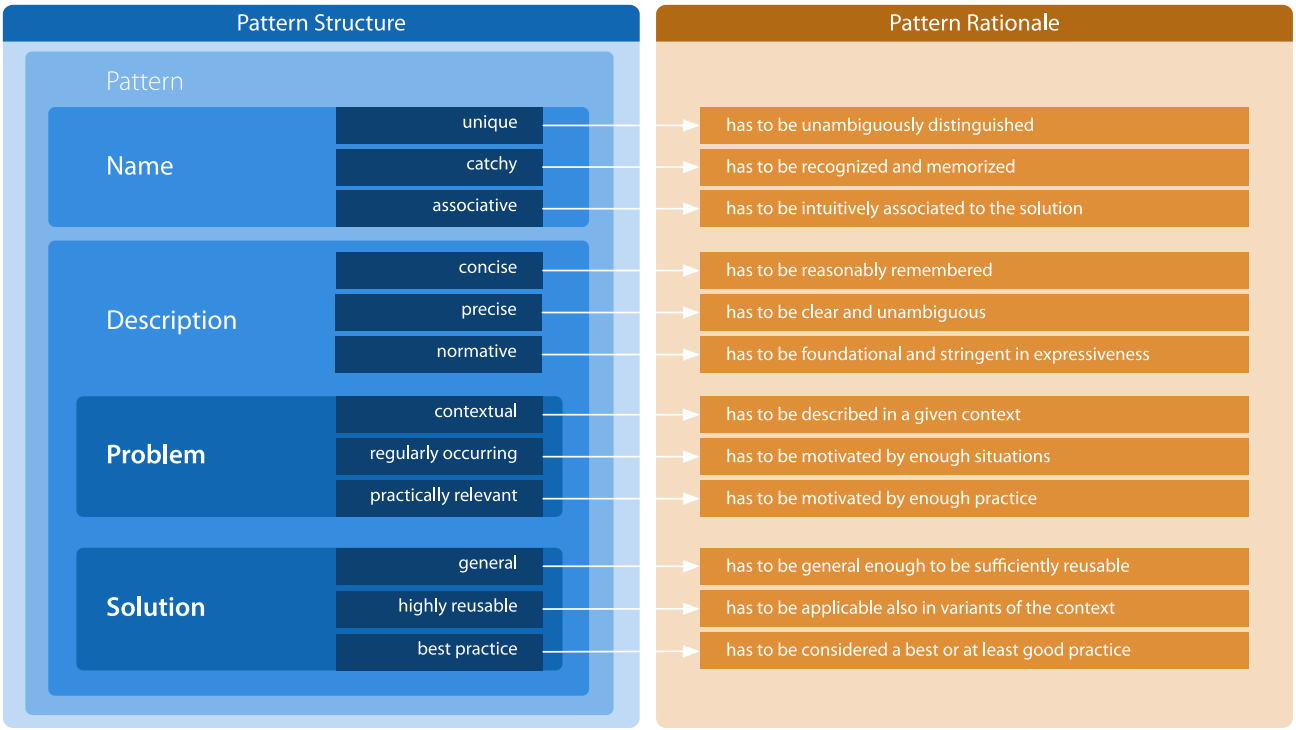
Fragen

- 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 Architecure)?
- Welche fünf Komponentenkategorien kennt man auf der Ebene der Micro Architecture (aka Programming)?

Architecture Patterns

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.



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 praxis-bewährten 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?

ARCHITECTURE FUNDAMENTALS

Layer Architectures

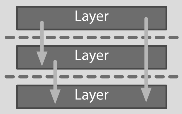
(Patterns)

TECHNISCHE UNIVERSITÄT MÜNCHEN

Layering Principle

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

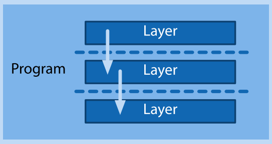
A Layer is not allowed to have relationships to or knowledge about any upper Layers. Additionally, for *Closed Layering*, each Layer is allowed to have relationships to and knowledge about the *directly* lower Layer only. In contrast to *Open Layering* or *Leaky Abstraction*, where each Layer is allowed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

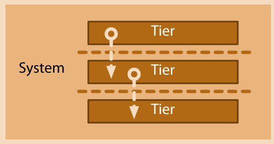
Rationale: Separation of Concern, Single Responsibility Principle, Mastering Complexity, Change Isolation, Functional Abstraction.



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.

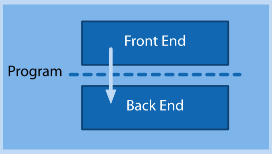
Rationale: Separation of Concern, Single Responsibility Principle, Mastering Complexity, Change Isolation, Functional Abstraction, Deployment Partitioning.



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.

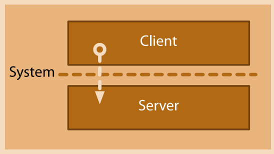
Rationale: Separation of Concern, Single Responsibility Principle, Mastering Complexity, Change Isolation, Functional Abstraction, Organisational Alignment.



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.

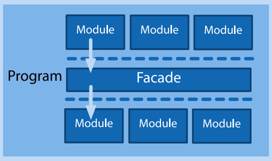
Rationale: Multi-User, User Computing Resource Leverage, Distributed Computing.



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.

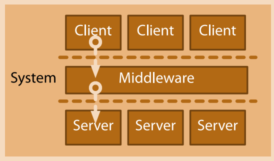
Rationale: Information Hiding, Cross-Cutting Concern Centralization, Functionality Orchestration, Authorization, Validation, Conversion.



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.



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. Zusätztlich 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**?