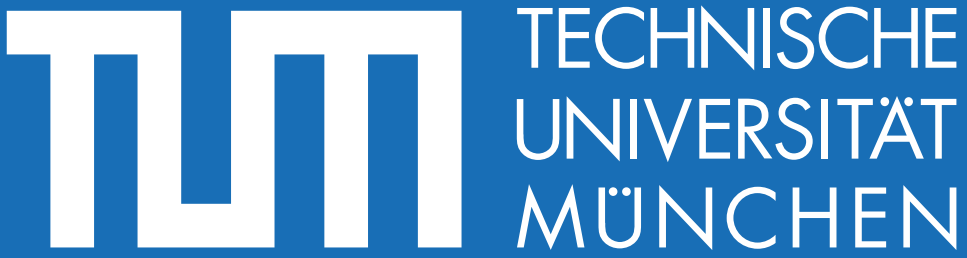
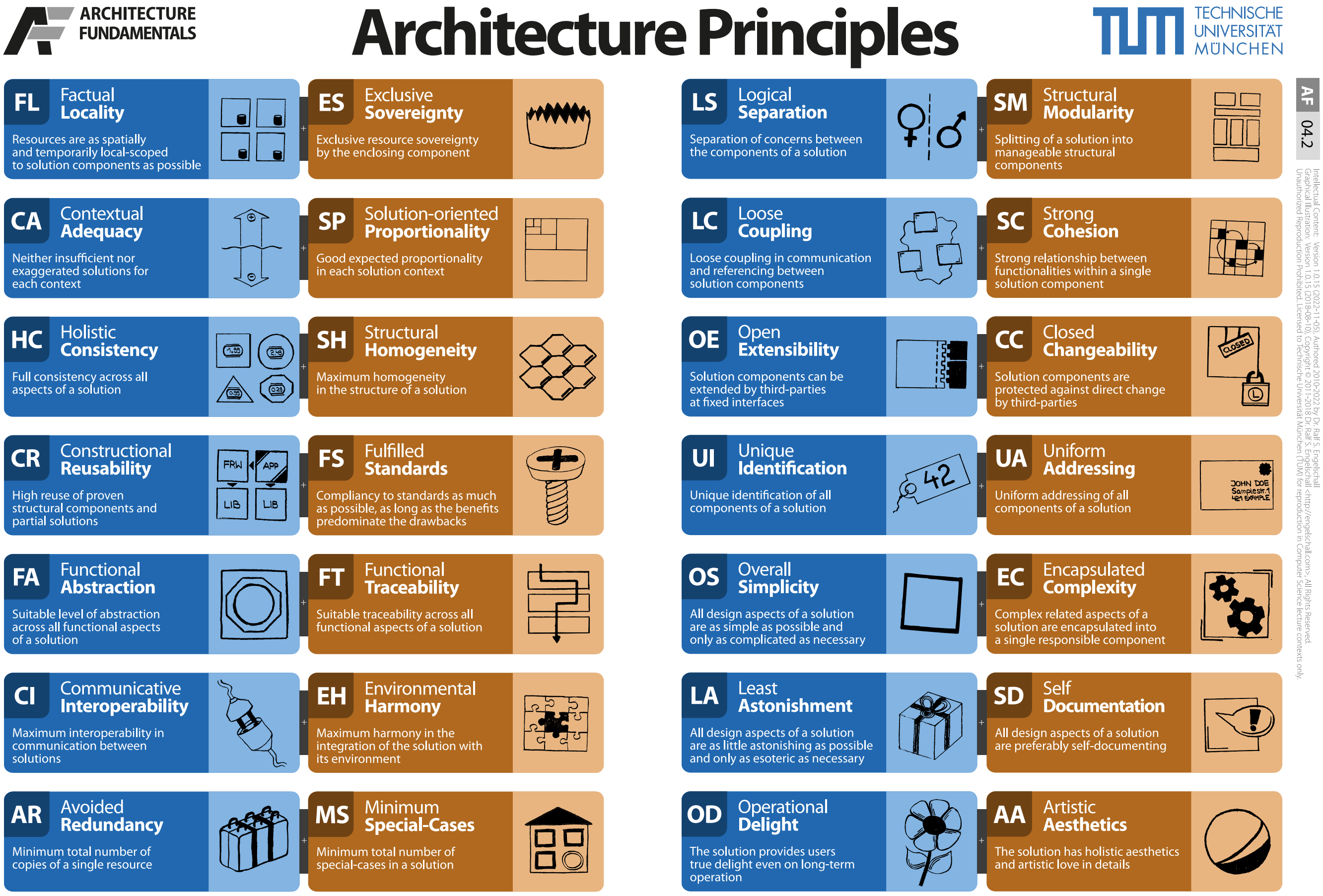


Software Engineering in Industrial Practice (SEIP)

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.

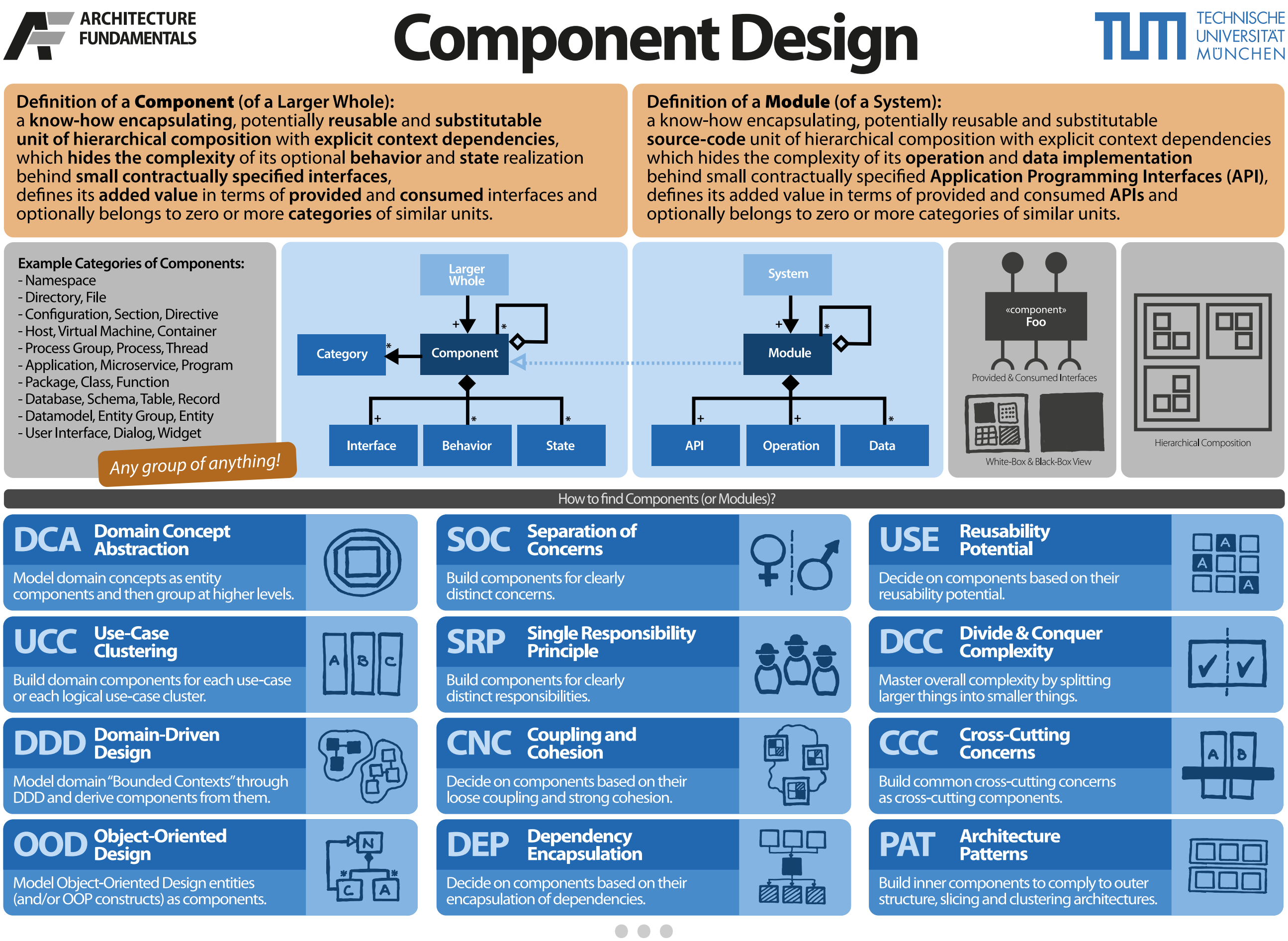
Beachte: das Prinzip **Logical Separation** (aka **Separation of Concern**) ist eines der wichtigsten, da aus ihm etliche andere Prinzipien fast automatisch folgen, unter anderem z.B. **Structural Modularity**.

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!


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!
Larger Whole
Category
Component
Interface
Behavior
State
System
Module
API
Operation
Data
«component» Foo
Provided & Consumed Interfaces
White-Box & Black-Box View
Hierarchical Composition
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.

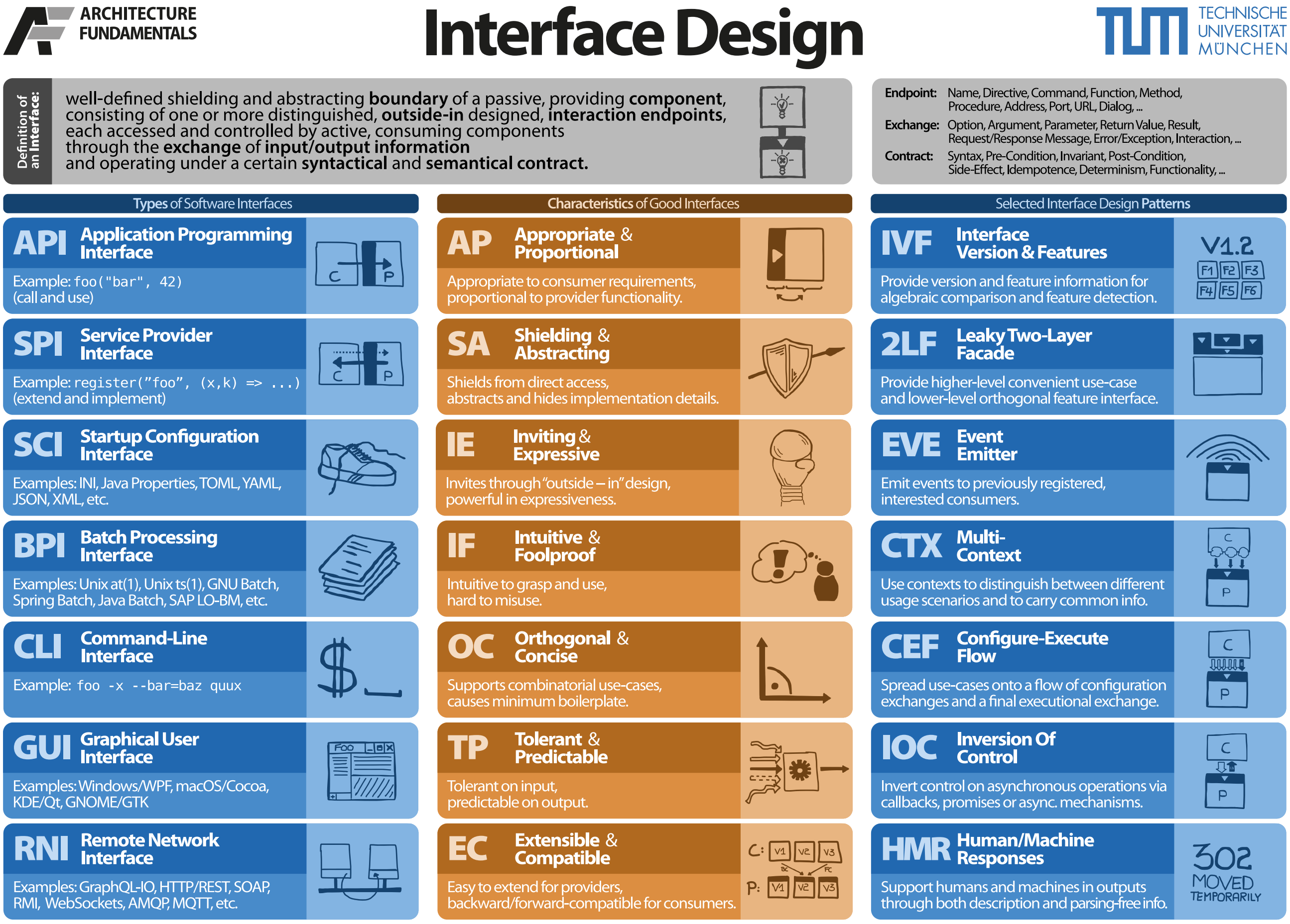
Eine Component **kapselt** ein bestimmtes **Know-How**, ist **potenziell wiederverwendbar** und **ersetzbar**. Eine Component besitzt ein **Verhalten** und einen **Zustand** und verbirgt die interne Komplexität von beidem hinter "kleinen" **vertraglichen Schnittstellen**. Sie stellt ihren Mehrwert über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


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.
V1.2
F1 F2 F3 F4 F5 F6
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.
302 MOVED TEMPORARILY

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.

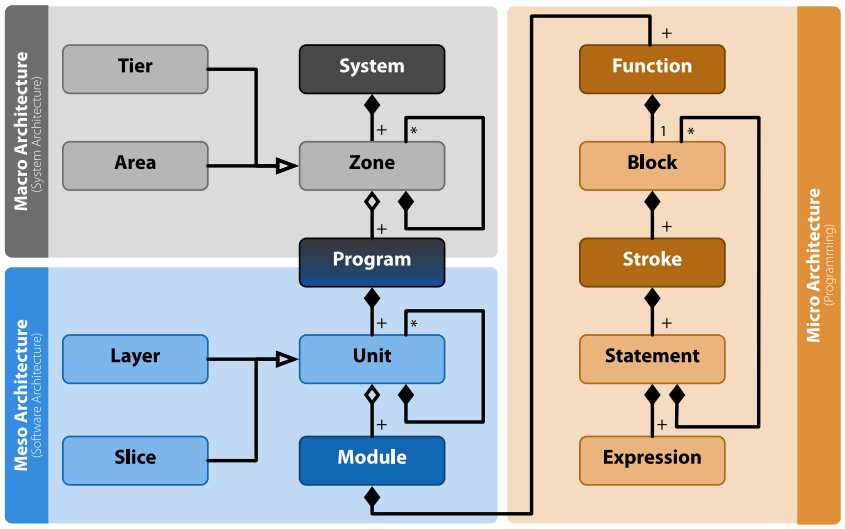
Es gibt zahlreiche Arten von Schnittstellen, die alle dieser Definition entsprechen. "Gute" Schnittstellen haben darüber hinaus bestimmte Eigenschaften/Charakteristiken. Die vier der besten Eigenschaften sind: **Proportional** (die Schnittstelle ist kleiner und in der Größe proportional zur dahinterliegenden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

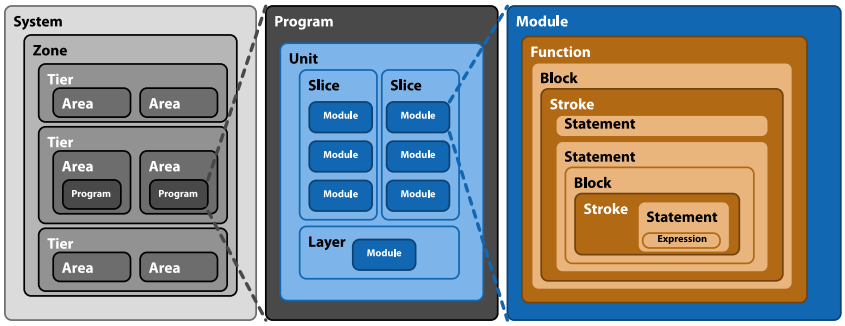
Es gibt zahlreiche Software Pattern für Schnittstellen. Ein interessantes Muster ist die **Leaky Two-Layer Facade**, bei dem eine Bibliothek zwei Schnittstellen besitzt: eine obere, bequeme und Use-Case-bezogene Schnittstelle und eine untere, orthogonale Feature-bezogene Schnittstelle. Der Witz ist, 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**.

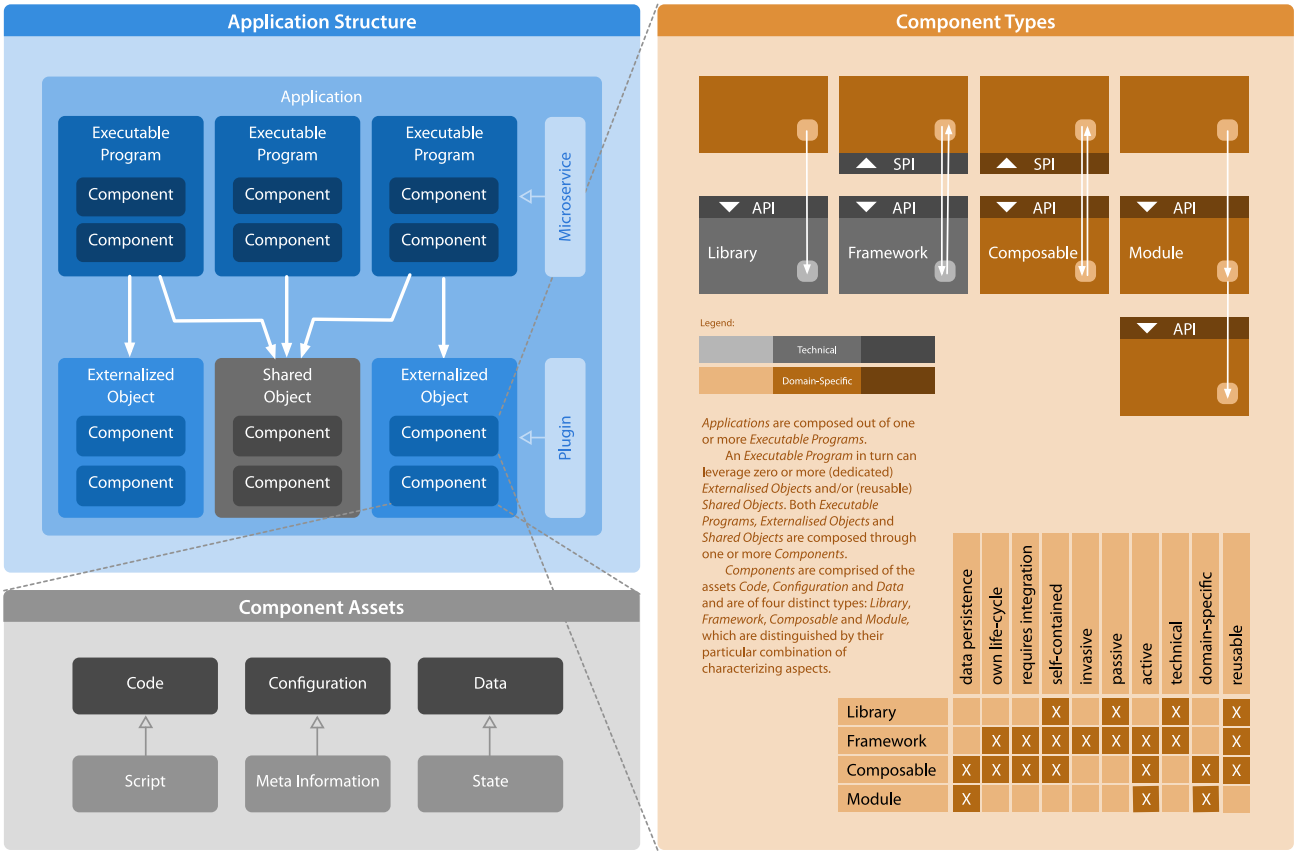
Die **Modules** bestehen in der Ebene Micro Architecture wiederum aus **Functions** und diese aus hierarchisch angeordneten (lexikalischen) **Blocks**, welche ihrerseits aus **Strokes** (aka "Thoughts") bestehen, welche ihrerseits aus **Statements** bestehen und diese bestehen am Ende aus einzelnen **Expressions**.

Die fünf **Primary Units of Thinking** sind **Systems**, **Programs**, **Modules**, **Functions** 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)?

Application Composition



Anwendungen werden aus einem oder mehreren Executable Programs zusammengesetzt. Ein Executable Program kann seinerseits null oder mehrere (dedizierte) Externalized Objects und/oder (wiederverwendbare) Shared Objects nutzen. Sowohl Executable Programs, Externalized Objects und Shared Objects bestehen aus einer oder mehreren Components. In einer Microservice Architecture werden die Executable Programs als Microservices bezeichnet. In einer Plugin Architecture werden die Externalized Objects als Plugins bezeichnet.

Es gibt vier verschiedene Typen von Components: Library (Bibliothek), Framework, Composable und Module (Modul). Sie lassen sich durch ihre besondere Kombination von charakteristischen Aspekten unterscheiden. Am wichtigsten ist, ob sie ein Application Programming Interface (API) für den Konsumenten der Component bereitstellen und/oder ob sie verlangen, dass der Konsument der Component eine Art von Service Provider Interface (SPI) erfüllen muß.

Fragen

- Was ist der wesentliche Unterschied zwischen einer Library und einem Framework?