































# **Software Engineering in der industriellen Praxis (SEIP)**

**Dr. Ralf S. Engelschall**

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

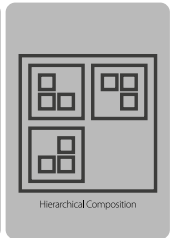
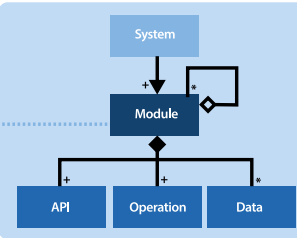
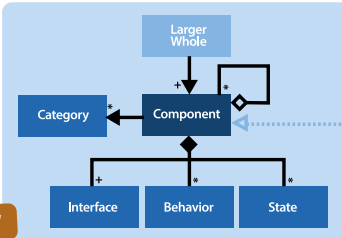
### Definition of a **Module** (of a System):

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)?

<b>DCA</b> Domain Concept Abstraction Model domain concepts as entity components and then group at higher levels. 	<b>SOC</b> Separation of Concerns Build components for clearly distinct concerns. 	<b>USE</b> Reusability Potential Decide on components based on their reusability potential. 
<b>UCC</b> Use-Case Clustering Build domain components for each use-case or each logical use-case cluster. 	<b>SRP</b> Single Responsibility Principle Build components for clearly distinct responsibilities. 	<b>DCC</b> Divide & Conquer Complexity Master overall complexity by splitting larger things into smaller things. 
<b>DDD</b> Domain-Driven Design Model domain "Bounded Contexts" through DDD and derive components from them. 	<b>CNC</b> Coupling and Cohesion Decide on components based on their loose coupling and strong cohesion. 	<b>CCC</b> Cross-Cutting Concerns Build common cross-cutting concerns as cross-cutting components. 
<b>OOD</b> Object-Oriented Design Model Object-Oriented Design entities (and/or OOP constructs) as components. 	<b>DEP</b> Dependency Encapsulation Decide on components based on their encapsulation of dependencies. 	<b>PAT</b> 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?

<b>Definition of an Interface:</b> well-defined shielding and abstracting <b>boundary</b> of a passive, providing <b>component</b> , consisting of one or more distinguished, <b>outside-in</b> designed, <b>interaction endpoints</b> , each accessed and controlled by active, consuming components through the <b>exchange</b> of <b>input/output</b> information and operating under a certain <b>syntactical</b> and <b>semantical</b> contract.				<b>Endpoint:</b> Name, Directive, Command, Function, Method, Procedure, Address, Port, URL, Dialog, ... <b>Exchange:</b> Option, Argument, Parameter, Return Value, Result, Request/Response Message, Error/Exception, Interaction, ... <b>Contract:</b> Syntax, Pre-Condition, Invariant, Post-Condition, Side-Effect, Idempotence, Determinism, Functionality, ...	
Types of Software Interfaces		Characteristics of Good Interfaces		Selected Interface Design Patterns	
<b>API</b> Application Programming Interface Example: foo("bar", 42) (call and use)				<b>IVF</b> Interface Version & Features <b>V1.2</b> Provide version and feature information for algebraic comparison and feature detection.	
<b>SPI</b> Service Provider Interface Example: register("foo", (x,k) => ...) (extend and implement)		<b>AP</b> Appropriate & Proportional Appropriate to consumer requirements, proportional to provider functionality.			
<b>SCI</b> Startup Configuration Interface Examples: INI, Java Properties, TOML, YAML, JSON, XML, etc.		<b>SA</b> Shielding & Abstracting Shields from direct access, abstracts and hides implementation details.		<b>2LF</b> Leaky Two-Layer Facade Provide higher-level convenient use-case and lower-level orthogonal feature interface.	
		<b>IE</b> Inviting & Expressive Invites through "outside-in" design, powerful in expressiveness.		<b>EVE</b> Event Emitter Emit events to previously registered, interested consumers.	
<b>BPI</b> Batch Processing Interface Examples: Unix at(1), Unix ts(1), GNU Batch, Spring Batch, Java Batch, SAP LO-BM, etc.		<b>IF</b> Intuitive & Foolproof Intuitive to grasp and use, hard to misuse.		<b>CTX</b> Multi-Context Use contexts to distinguish between different usage scenarios and to carry common info.	
		<b>OC</b> Orthogonal & Concise Supports combinatorial use-cases, causes minimum boilerplate.		<b>CEF</b> Configure-Execute Flow Spread use-cases onto a flow of configuration exchanges and a final executional exchange.	
<b>CLI</b> Command-Line Interface Example: foo -x --bar=baz quux				<b>IOC</b> Inversion Of Control Invert control on asynchronous operations via callbacks, promises or async. mechanisms.	
<b>GUI</b> Graphical User Interface Examples: Windows/WPF, macOS/Cocoa, KDE/Qt, GNOME/GTK		<b>TP</b> Tolerant & Predictable Tolerant on input, predictable on output.			
<b>RNI</b> Remote Network Interface Examples: GraphQL/HTTP, REST, SOAP, RMI, WebSockets, AMQP, MQTT, etc.		<b>EC</b> Extensible & Compatible Easy to extend for providers, backward/forward-compatible for consumers.		<b>HMR</b> Human/Machine Responses <b>302 MOVED TEMPORARILY</b> 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.

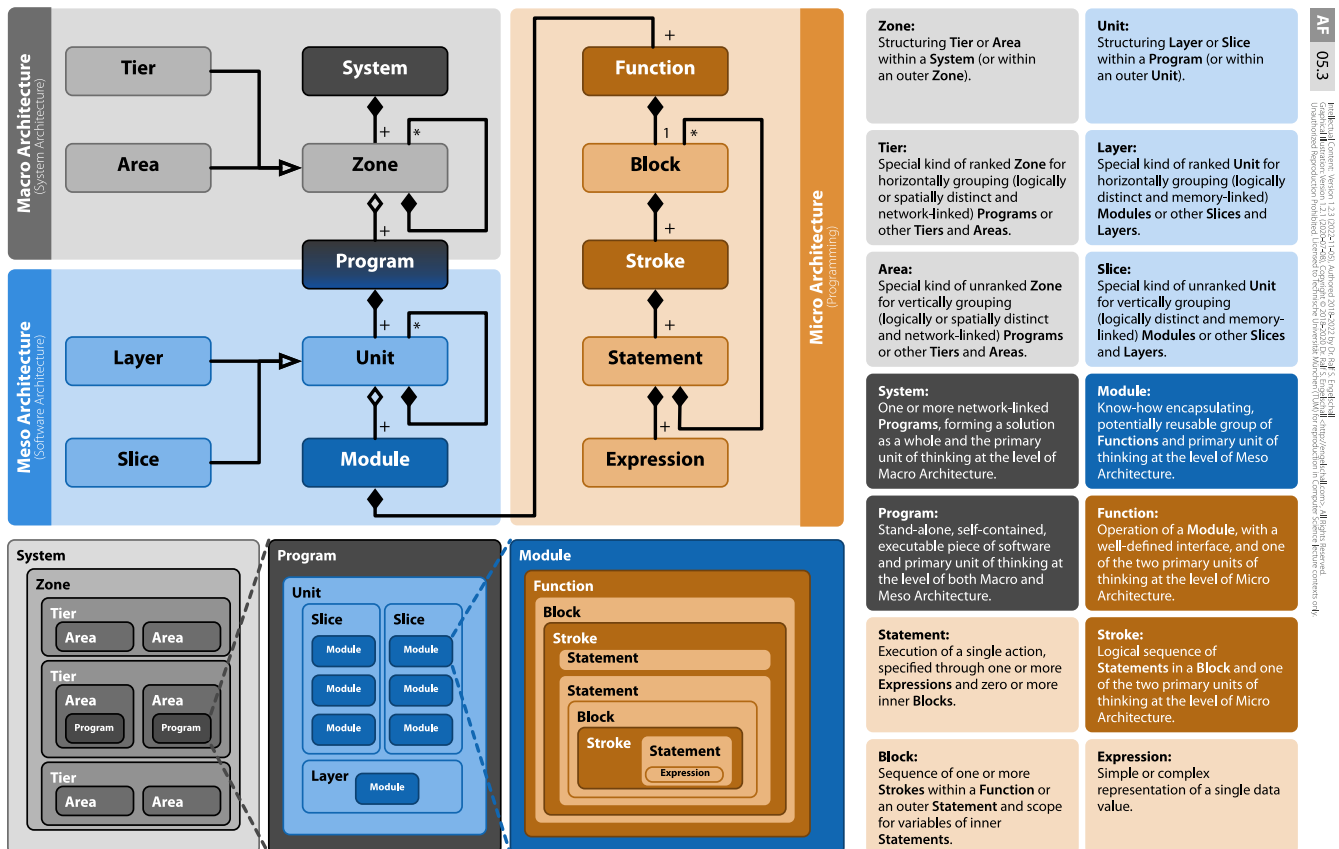
Auf jeden Interaktionsendpunkt wird dabei von einer aktiven konsumierenden Komponente durch den Austausch (**exchange**) von **input/output** Informationen (**information**) zugegriffen und wird unter einem bestimmten syntaktischen (**syntactical**) und semantischen (**semantical**) Vertrag (**contract**)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

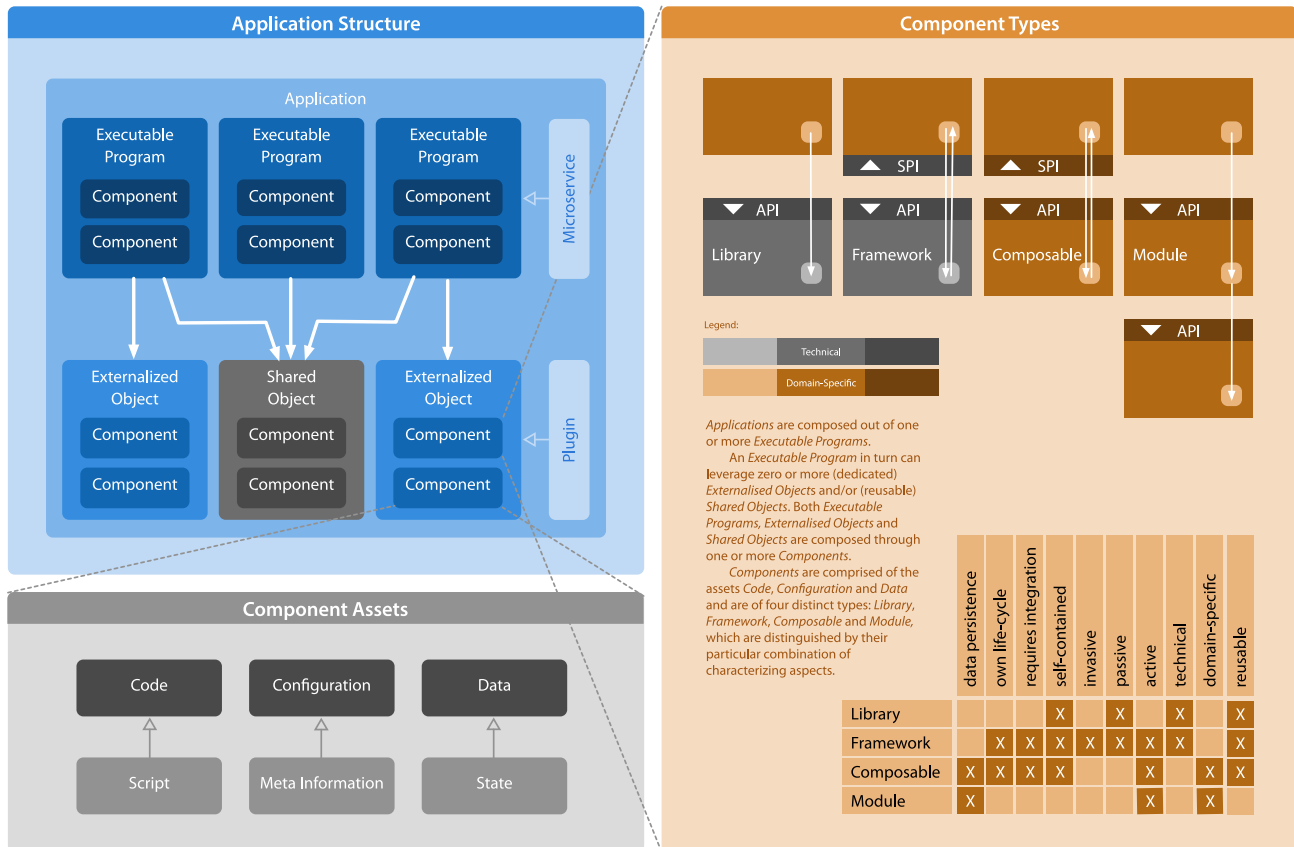
Diese **Programs** bestehen in der Ebene Meso Architecture wiederum aus hierarchisch angeordneten **Units**, welche entweder (horizontale) **Layer** oder (vertikale) **Slices** sein können. Die **Units** bestehen ihrerseits aus **Modules**.

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



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?