

Software Engineering in Industrial Practice (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 Constructional 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.

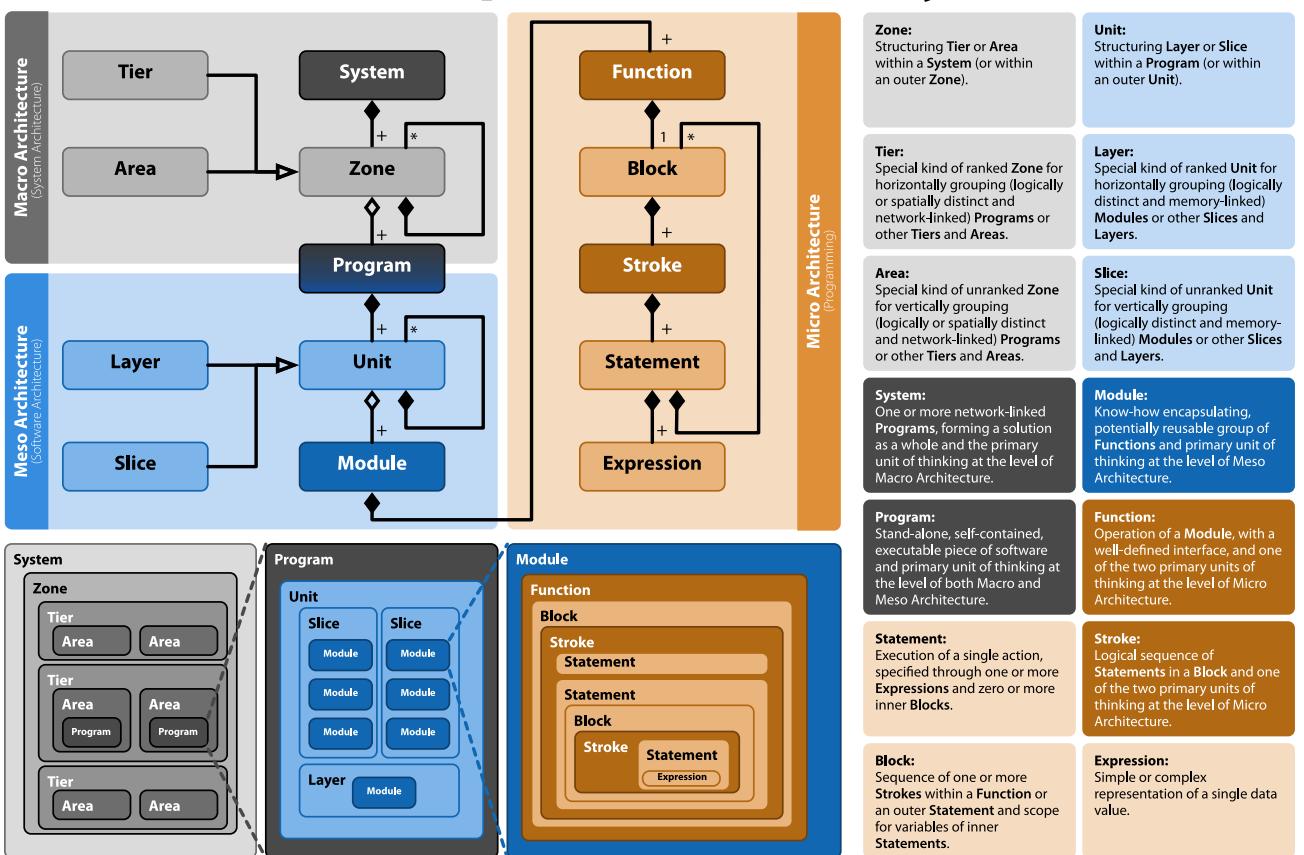
Fragen

- ?

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

Definition of an Interface:		<p>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.</p>		
Types of Software Interfaces		Characteristics of Good Interfaces		
API Application Programming Interface	Example: foo ("bar", 42) (call and use)		AP Appropriate & Proportional	Appropriate to consumer requirements, proportional to provider functionality.
SPI Service Provider Interface	Example: register("foo", (x, k) => ...) (extend and implement)		SA Shielding & Abstracting	Shields from direct access, abstracts and hides implementation details.
SCI Startup Configuration Interface	Examples: INI, Java Properties, TOML, YAML, JSON, XML, etc.		IE Inviting & Expressive	Invites through 'outside-in' design, powerful in expressiveness.
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.
CLI Command-Line Interface	Example: foo -x --bar=baz quux		OC Orthogonal & Concise	Supports combinatorial use-cases, causes minimum boilerplate.
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-O, HTTP/REST, SOAP, RMI, WebSockets, AMQP, MQTT, etc.		EC Extensible & Compatible	Easy to extend for providers, backward/forward-compatible for consumers.
		<img alt="Icon		

Component Hierarchy



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**.

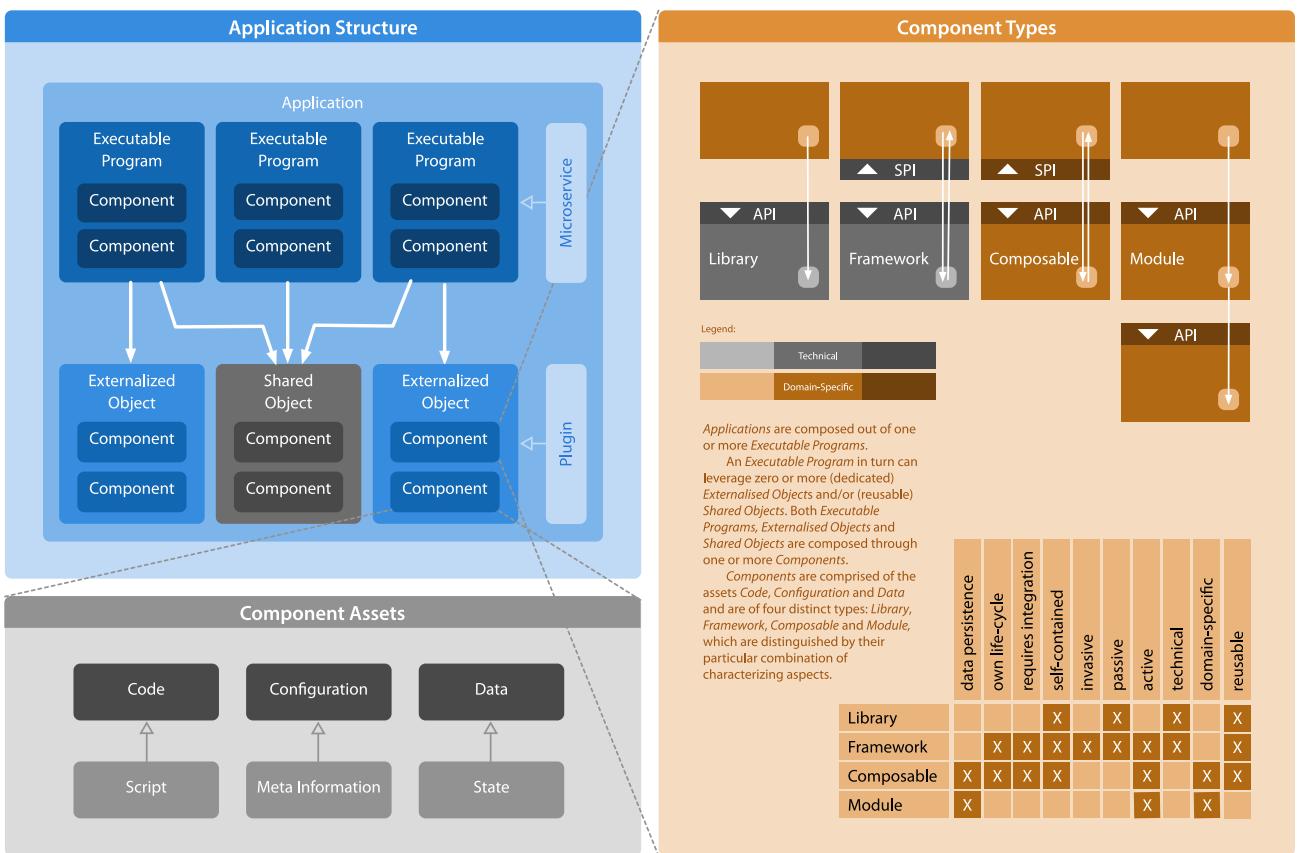
Diese **Programs** bestehen in der Ebene Meso Architecture wiederum aus hierarchisch angeordneten **Units**, welche entweder (horizontale) **Layers** oder (vertikale) **Slices** sein können. Die **Units** bestehen ihrerseits aus **Modules**.

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?