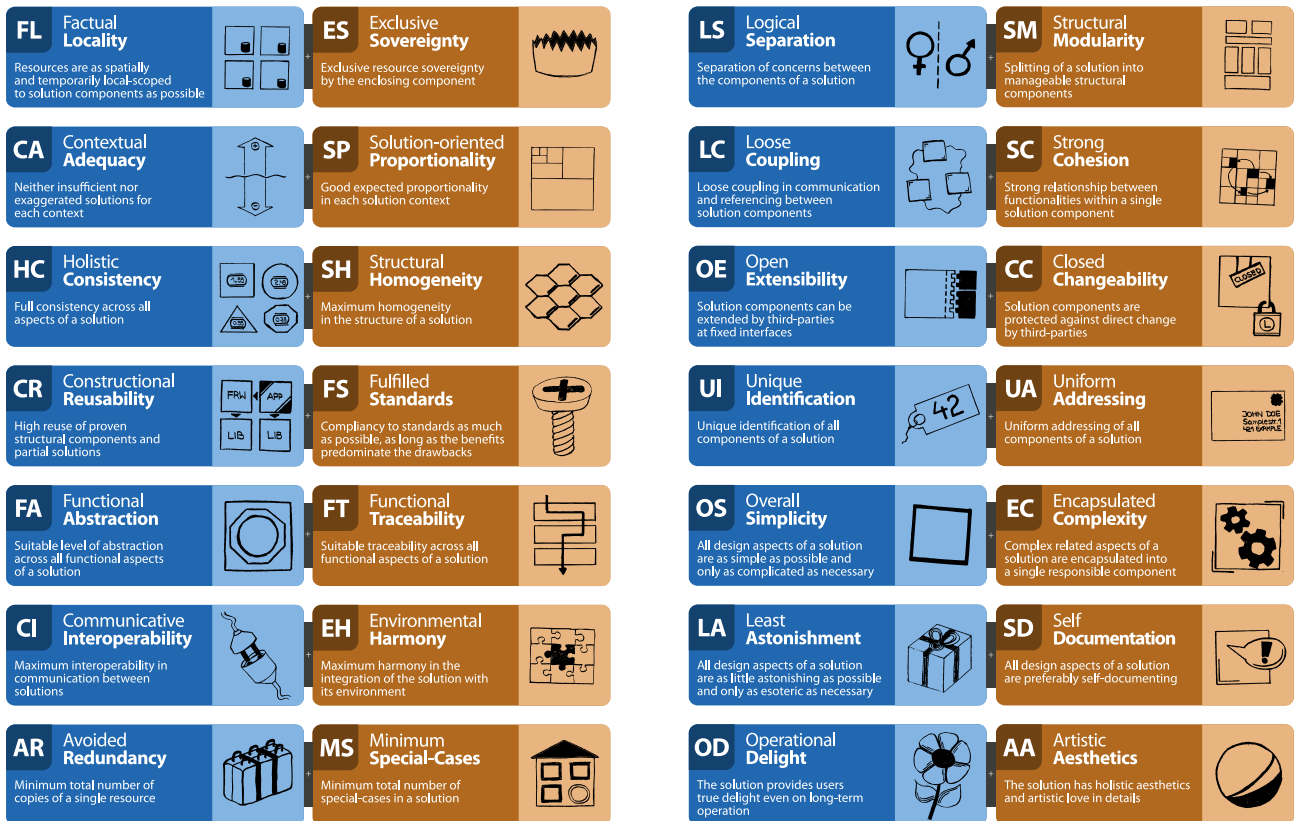




# **Software Engineering in Industrial Practice (SEIP)**

Dr. Ralf S. Engelschall



In IT Architecture, one follows **Architecture Principles**, which summarize basic principles and procedures. One knows 28 principles that can be grouped into 14 pairs since always two principles are very close regarding the content. The architect should follow the principles in general, but he may violate them as long as he has a good reason for it. The best reason would be a particular project-specific requirement.

Note: The principle **Logical Separation** (aka **Separation of Concern**) is one of the most important, since from it several other principles almost automatically follow, including, e.g., **Structural Modularity**.

Note: The principles **Loose Coupling** and **Strong Cohesion** are known in the literature as the combined principle “Loose Coupling, Strong Cohesion.” The principles **Open Extensibility** and **Closed Changeability** are known in the literature as the combined principle “Open-Close.”

Note: The principle **Overall Simplicity** is one of the hardest to implement because nothing in IT is really easy. Everything only looks simple as long as one does not have enough understanding about it. After that, one first has to make it “simple” painstakingly. That’s the art of architecture: simplify difficult things! If something cannot be simplified further and still has a certain complexity, following the principle **Encapsulated Complexity**, one at least can try to shadow it.

## Questions

- 🔍 List at least 4 essential **Architecture Principles**!

## 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.

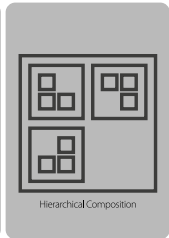
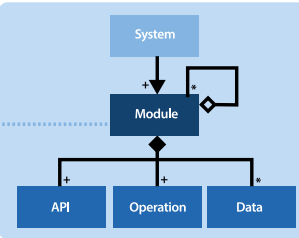
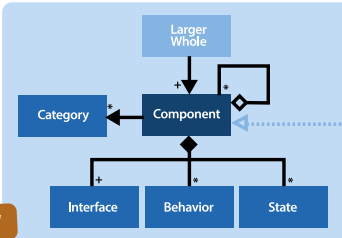
## Definition of a **Module** (of a System):

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.	

Software Architecture is all about **Components** and **Interfaces**. Therefore, **Component Design** is a central task of the architect.

A component **encapsulates** a certain **know-how**, is **potentially reusable** and **replaceable**. A component has a **behaviour** and a **state** and hides the internal complexity of both behind "small" **contractual interfaces**. It provides its added value through the difference between provided and consumed interfaces. It can be considered as a **Whitebox** or as a **Blackbox**, depending on whether the internal details can be viewed from outside or not. Components are arranged hierarchically, may belong to specific **categories** and have **explicit dependencies** among each other.

A distinction is made between the more general concept of **Component** ("any group of anything") and the more specific concept of the (via Source code defined) **Module**.

Components can be found in many different ways. Most of them are directly derived from existing methods, principles, or patterns. The two most important ways for a component cut in practice are: **Separation of Concerns** (which unique concern or task has the component?) and **Single Responsibility Principle** (what is the unique responsibility of the component?).

## Questions

- ❓ List at least 7 properties/aspects which a Component has!
- ❓ What are the two most important ways to find a component cut in practice?

<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 information</b> and operating under a certain <b>syntactical</b> and <b>semantical contract</b> .		<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	
<b>API</b> Application Programming Interface Example: foo("bar", 42) (call and use)		<b>AP</b> Appropriate & Proportional Appropriate to consumer requirements, proportional to provider functionality.	
<b>SPI</b> Service Provider Interface Example: register("foo", (x,k) => ...) (extend and implement)		<b>SA</b> Shielding & Abstracting Shields from direct access, abstracts and hides implementation details.	
<b>SCI</b> Startup Configuration Interface Examples: INI, Java Properties, TOML, YAML, JSON, XML, etc.		<b>IE</b> Inviting & Expressive Invites through "outside-in" design, powerful in expressiveness.	
<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>CLI</b> Command-Line Interface Example: foo -x --bar=baz quux		<b>OC</b> Orthogonal & Concise Supports combinatorial use-cases, causes minimum boilerplate.	
<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.	
		Selected Interface Design Patterns	
		<b>IVF</b> Interface Version & Features Provide version and feature information for algebraic comparison and feature detection.	<b>V1.2</b> 
		<b>2LF</b> Leaky Two-Layer Facade Provide higher-level convenient use-case and lower-level orthogonal feature interface.	
		<b>EVE</b> Event Emitter Emit events to previously registered, interested consumers.	
		<b>CTX</b> Multi-Context Use contexts to distinguish between different usage scenarios and to carry common info.	
		<b>CEF</b> Configure-Execute Flow Spread use-cases onto a flow of configuration exchanges and a final executional exchange.	
		<b>IOC</b> Inversion Of Control Invert control on asynchronous operations via callbacks, promises or async. mechanisms.	
		<b>HMR</b> Human/Machine Responses Support humans and machines in outputs through both description and parsing-free info.	<b>302</b> MOVED TEMPORARILY

An **interface** is a **well-defined, shielding, abstracting boundary** of a passive providing **component**, which consists of one or more clearly distinguishable **interaction endpoints**.

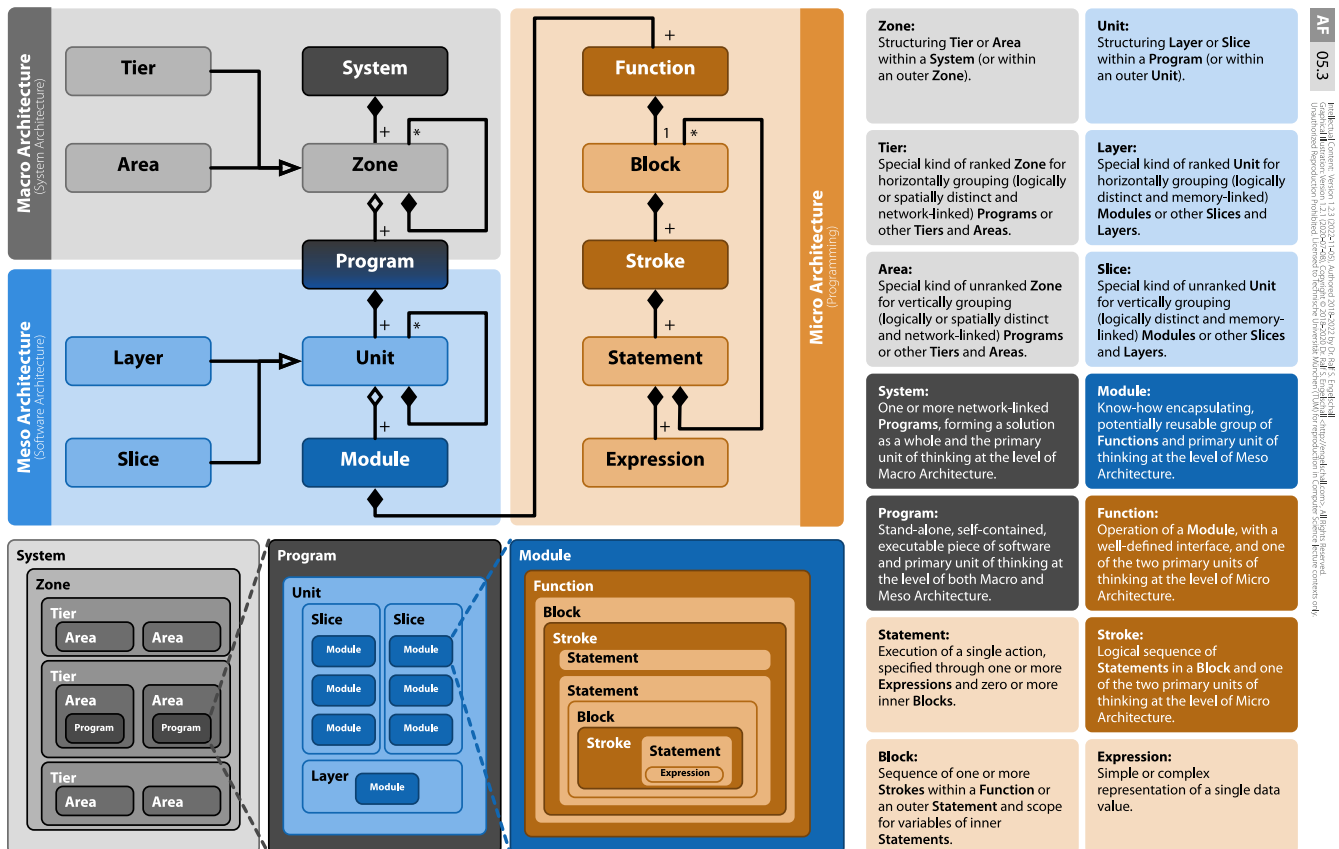
At each interaction endpoint, an active, consuming component is accessed through the **exchange** of **input/output information** and is operated under a specific **syntactical** and **semantical contract**.

There are numerous kinds of interfaces, all of which meet this definition. In addition, "good" interfaces have specific Properties/Characteristics. The four of the best properties are: **Proportional** (the interface is smaller and in size proportional to the functionality behind it), **Expressive** (the interface provides a powerful programming model), **Orthogonal** (the interface allows combinatorial Use-Cases), and **Concise** (the interface generates little "Boilerplate Code" during use).

There are numerous software patterns for interfaces. An interesting pattern is the **Leaky Two-Layer Facade**, in which a library has two interfaces: an upper, convenient, and Use-Case-related interface and a lower, orthogonal Feature-related interface. The trick is that the upper interface is implemented by the lower interface only and that the lower interface "shines through" ("leaky" or Open Layering).

## Questions

- ? List at least 8 properties/aspects which define an **Interface**!
- ? List at least 4 properties/characteristics of **good Interfaces**!



A **Component** is “any group of anything” in Software Architecture. Nevertheless, there are prominent component categories that form an particular, omnipresent **Component Hierarchy** in Software Engineering. This consists of the three levels **Macro Architecture** (aka System Architecture), **Meso Architecture** (aka Software Architecture) and **Micro Architecture** (aka Programming).

In the Macro Architecture level, one has to deal with **Systems** (aka Applications) which consist of hierarchically arranged infrastructural **Zones**, which can be either (horizontal) **Tiers** or (vertical) **Areas**. The **Zones** themselves consist of **Programs**.

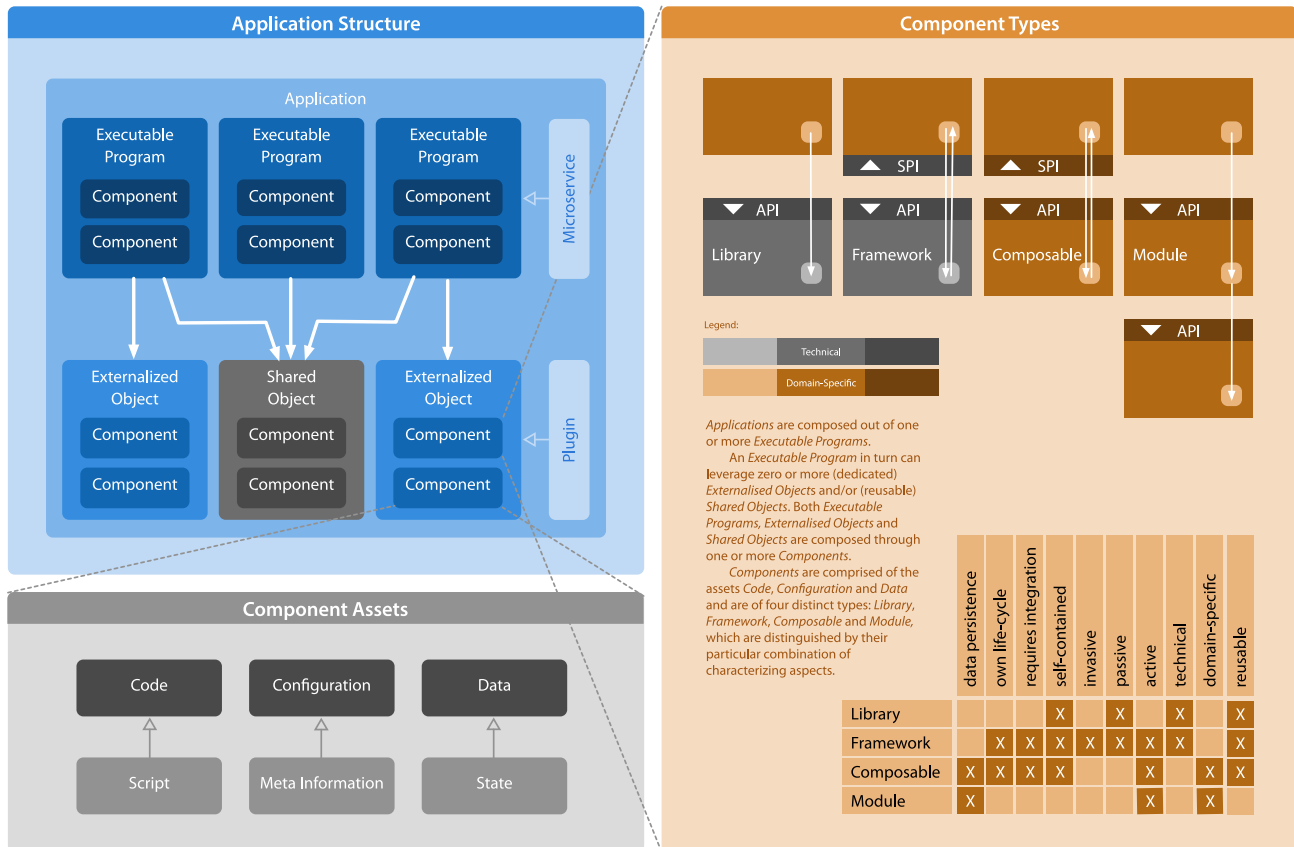
These **Programs**, at the level of the Meso Architecture, consist of hierarchically arranged **Units**, which can be either (horizontal) **Layers** or (vertical) **Slices**. The **Units** themselves consist of **Modules**.

The **Modules**, at the level of the Micro Architecture, consist of **Functions** and these consist of hierarchically arranged (lexical) **Blocks**, which in turn consist of **Strokes** (aka “Thoughts”), which in turn consist of **Statements** and these at the end consist of individual **Expressions**.

The five **Primary Units of Thinking** are **Systems, Programs, Modules, Functions** and **Strokes**.

## Questions

- ❓ Which three component categories are known at the level of Macro Architecture (aka System Architecture)?
- ❓ Which three component categories are known at the level of Meso Architecture (aka Software Architecture)?
- ❓ Which five component categories are known at the level of Micro Architecture (aka Programming)?



Applications are composed out of one or more Executable Programs. An Executable Program in turn can leverage zero or more (dedicated) Externalised Objects and/or (reusable) Shared Objects. Both Executable Programs, Externalised Objects and Shared Objects are composed through one or more Components. In a Microservice Architecture, the Executable Programs are called Microservices. In a Plugin Architecture, the Externalised Objects are called Plugins.

There are four distinct types of Components: Library, Framework, Composable and Module. They can be distinguished by their particular combination of characterizing aspects. Most prominently, whether they provide an Application Programming Interface (API) to the consumer of the Component and/or whether they require the consumer of the Component to fulfill some sort of Service Provider Interface (SPI).

## Questions

- ? What is the main difference between a Library and a Framework?