TECHNISCHE UNIVERSITÄT MÜNCHEN

Software Engineering in der industriellen Praxis (SEIP)

Dr. Ralf S. Engelschall

Architecture Principles

FL Factual **Locality**
Resources are as spatially and temporarily local-scoped to solution components as possible

ES Exclusive **Sovereignty**
Exclusive resource sovereignty by the enclosing component

CA Contextual **Adequacy**
Neither insufficient nor exaggerated solutions for each context

SP Solution-oriented **Proportionality**
Good expected proportionality in each solution context

HC Holistic **Consistency**
Full consistency across all aspects of a solution

SH Structural **Homogeneity**
Maximum homogeneity in the structure of a solution

CR Constructional **Reusability**
High reuse of proven structural components and partial solutions

FS Fulfilled **Standards**
Compliancy to standards as much as possible, as long as the benefits predominate the drawbacks

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

CI Communicative **Interoperability**
Maximum interoperability in communication between solutions

EH Environmental **Harmony**
Maximum harmony in the integration of the solution with its environment

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

LS Logical **Separation**
Separation of concerns between the components of a solution

SM Structural **Modularity**
Splitting of a solution into manageable structural components

LC Loose **Coupling**
Loose coupling in communication and referencing between solution components

SC Strong **Cohesion**
Strong relationship between functionalities within a single solution component

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

UI Unique **Identification**
Unique identification of all components of a solution

UA Uniform **Addressing**
Uniform addressing of all components 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

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

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 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**!

ARCHITECTURE FUNDAMENTALS

Component Design

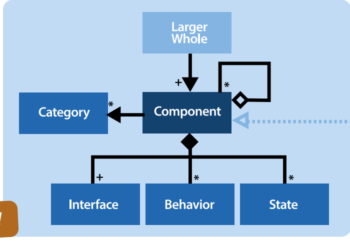
TECHNISCHE UNIVERSITÄT MÜNCHEN

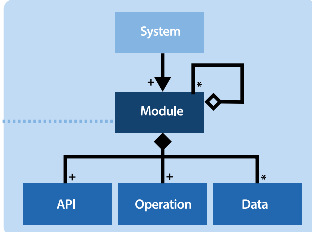
Definition of a Component (of a Larger Whole):
a **know-how encapsulating**, potentially **reusable** and **substitutable unit of hierarchical composition** with **explicit context dependencies**, which **hides the complexity** of its optional **behavior** and **state** realization behind **small contractually specified interfaces**, defines its **added value** in terms of **provided** and **consumed** interfaces and optionally belongs to zero or more **categories** of similar units.

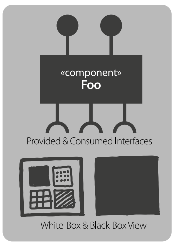
Definition of a Module (of a System):
a know-how encapsulating, potentially reusable and substitutable **source-code** unit of hierarchical composition with explicit context dependencies which hides the complexity of its **operation** and **data implementation** behind small contractually specified **Application 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!

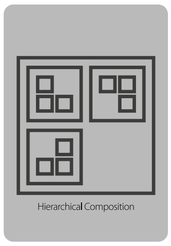






Provided & Consumed Interfaces

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.

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

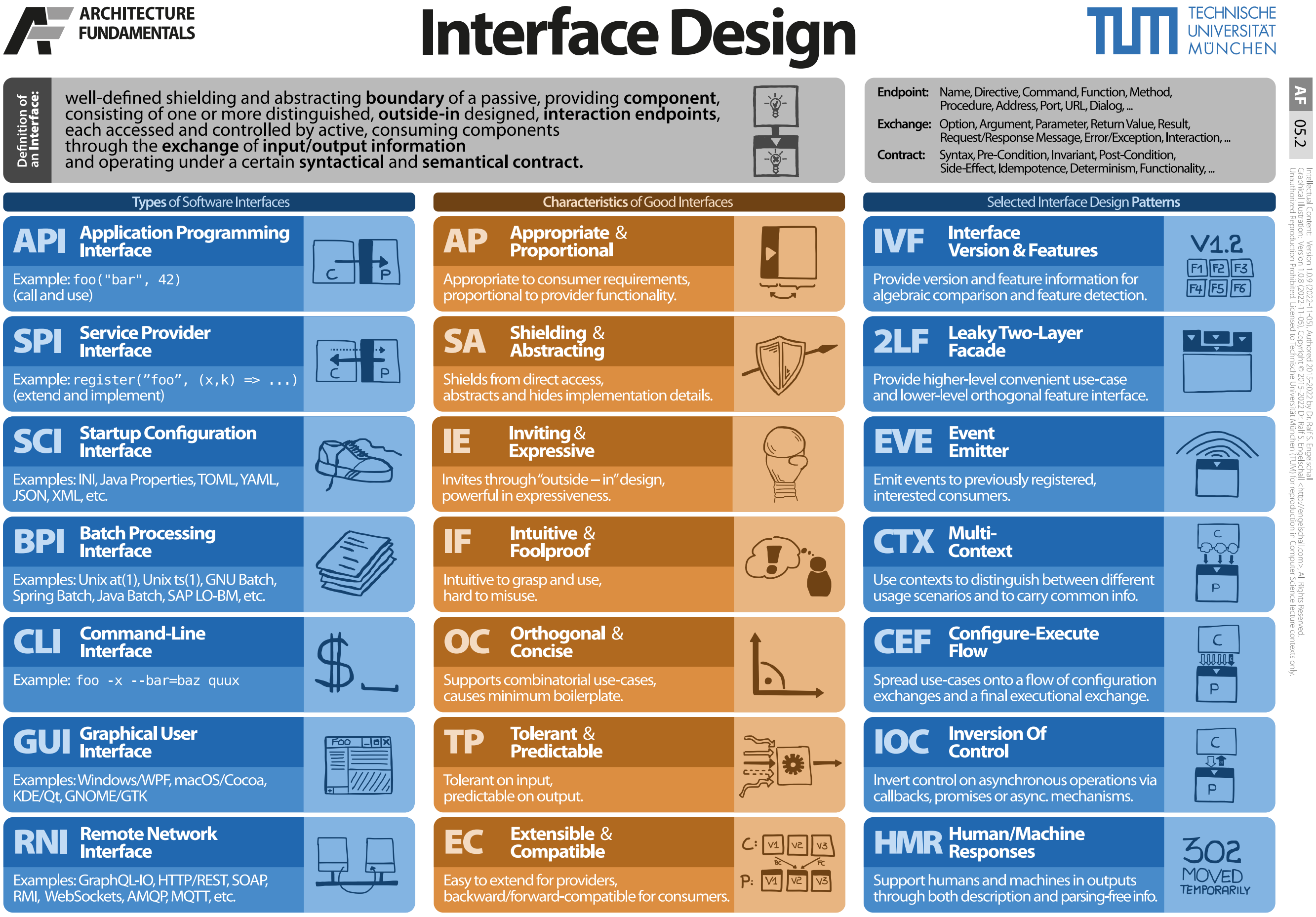
A component **encapsulates** a certain **know-how**, is **potentially reusable** and **replaceable**. A component has a **behaviour** and a **state** and hides the internal complexity of both behind "small" **contractual interfaces**. It provides its added value through the difference between provided 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?



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**.

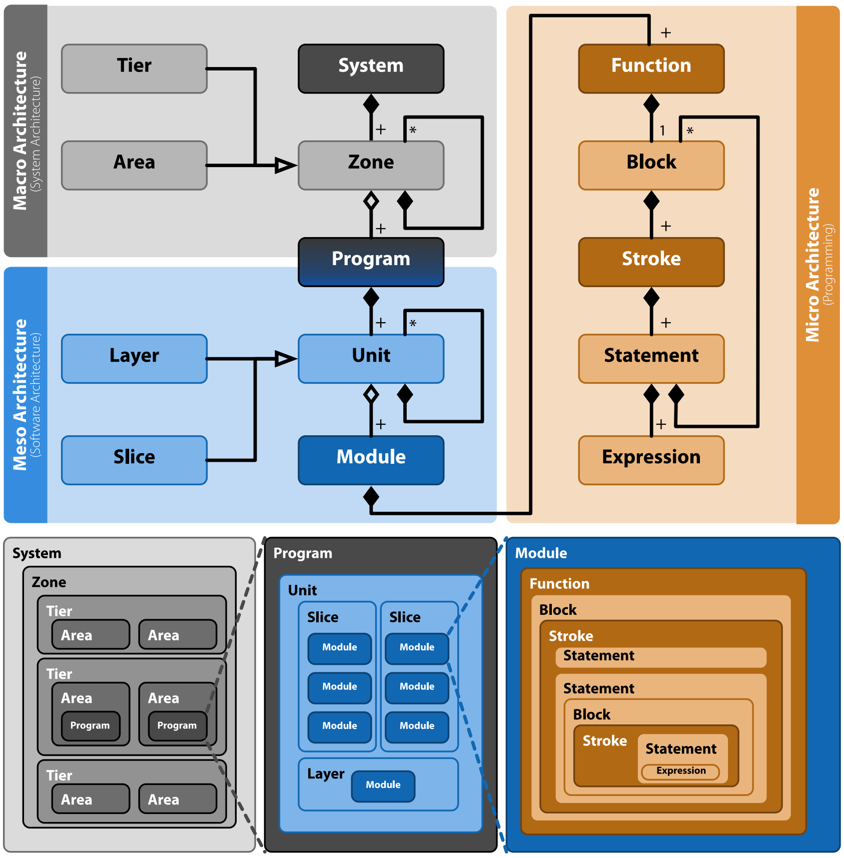
There are numerous kinds of interfaces, all of which meet this definition. In addition, "good" interfaces have specific Properties/Characteristics. The four of the best properties are: **Proportional** (the interface is smaller and in size proportional to the functionality behind it), **Expressive** (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**!

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.

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**.

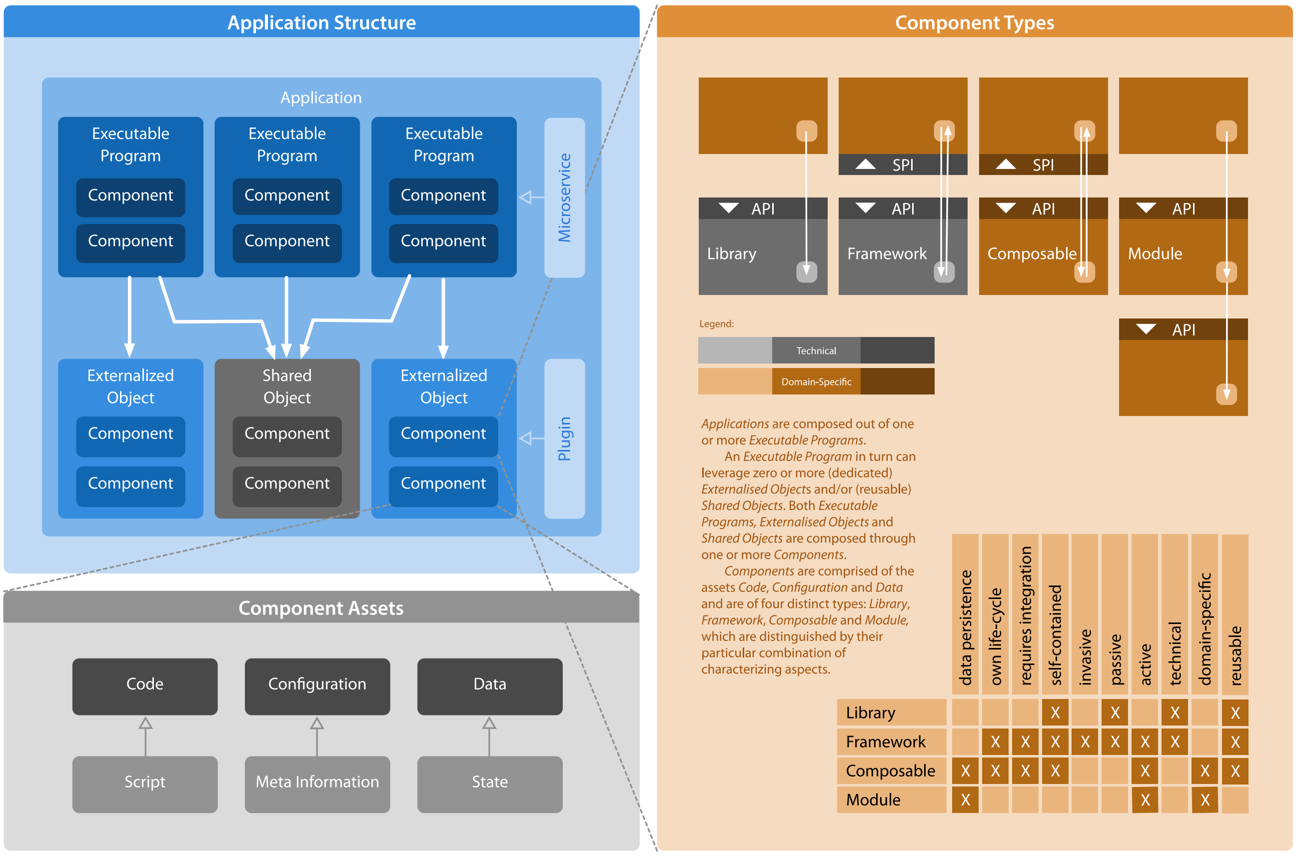
The **Modules**, at the level of the Micro Architecture, consist of **Functions** and these consist of hierarchically arranged (lexical) **Blocks**, which in turn consist of **Strokes** (aka “Thoughts”), which in turn consist of **Statements** and these at the end consist of individual **Expressions**.

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)?

Application Composition



	data persistence	own life-cycle	requires integration	self-contained	invasive	passive	active	technical	domain-specific	reusable
Library				X		X		X		X
Framework		X	X	X	X	X	X	X		X
Composable	X	X	X	X			X		X	X
Module	X						X		X	

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?