iSAQB® Glossary of Software Architecture Terminology

2025.1-rev2-EN-20250117

Table of Contents

Introduction

This book contains a glossary of *software architecture terminology*.

It can aid in preparation for the iSAQB® e. V. examination *Certified Professional for Software Architecture - Foundation Level*©.

Please be aware: This glossary is **not** intended to be a primer or course book on software architecture, but just a collection of definitions and links to further information.

Furthermore, you find proposals for translations of the iSAQB® terminology, currently between English and German (and vice-versa).

Finally, this book contains numerous references to books and other resources, many of which we quoted in the definitions.

This book is work in progress.

Errors or omissions can also be reported in our issue tracker on GitHub, where the authors maintain the original sources for this book.

Personal Comments

Several of the terms contained in this book have been commented by one or several authors:

Comment (Gernot Starke)

Some terms might be especially important, or sometimes there are some subtle aspects involved. Comments like these give a personal opinion and do **not** necessarily reflect the iSAQB®.

Terms Can Be Referenced

All terms in the glossary have unique URLs to the (free) online version of the book therefore they can be universally referenced, both from online- and print documentation.

Our URL scheme is quite simple:

- The base URL is `https://public.isaqb.org/glossary/glossary-en.html`
- We just add the prefix `#term-` in front of the term to be referenced, then the term itself, with hyphens ("-") instead of blanks.

For example our description of the term *software architecture* can be referenced (hyperlinked) with `https://public.isaqb.org/glossary/glossary-en.html#term-software-architecture`

Nearly all terms are hyperlinked with their full names, with very few examples that are referenced by their (common) abbreviations, like UML or DDD.

License

This book is licensed under a Creative Commons Attribution 4.0 International License. The following is only a brief summary and no substitution for the real license.

The **CC BY 4.0** license means that you might:

- Share — copy and redistribute the material in any medium or format
- Adapt — remix, transform, and build upon the material for any purpose, even commercially.
- The licensor cannot revoke these freedoms as long as you follow the license terms.

You must:

- Give appropriate credit,
- Provide a link to the license (https://creativecommons.org/licenses/by/4.0/)), and
- Indicate if (and which) changes were made with respect to the original.

Acknowledgements

Several parts of this glossary have been contributed by the following volunteers and sponsors (apart from the numerous authors.)

- The definitions of about 120 terms have been donated by Gernot Starke, originally compiled for one of his books.
- A number of definitions in context of system improvement and evolution was contributed by the aim42 open source project.

Contributing

Contributions are welcome

In case find errors, omissions or typos, or want to contribute additional content - please feel free to do this via one of the following ways:

1. Open an issue in our GitHub repository
2. Fork the repository and create a pull request.
3. Write an email to the authors,

Your input is highly appreciated by the authors.

Terms

A

Abstraction

The process of removing details to focus attention on aspects of greater importance. Similar in nature to the process of generalization.

A view of an element that focuses on the information relevant to a particular purpose, ignoring additional or other information.

A design construct as in "Building blocks should depend on abstractions rather than on implementations."

Abstractness

Metric for the source code of object oriented systems: The number of abstract types (interfaces and abstract classes) divided by the total number of types.

Accessibility Quality Attribute

Degree to which a product or system can be used by people with the widest range of characteristics and capabilities to achieve a specified goal in a specified context of use. Is a sub-characteristic of: usability.

Refer to [ISO-25010].

Accountability Quality Attribute

Degree to which the actions of an entity can be traced uniquely to the entity. Is a sub-characteristic of: security.

Refer to [ISO-25010].

Accreditation

Determination procedure and certification by an authorised accreditation body (here the iSAQB®) confirming that the applicant meets the organizational, technical and qualitative requirements as a training provider.

Accreditation Body

The application for accreditation must be submitted through the *accreditation body* designated by the iSAQB. The accreditation body is the contact point for all questions of the training provider during the accreditation. It coordinates the accreditation procedure, carries out the formal assessment of the documents submitted and organises the technical assessment in the AUDIT WORKING GROUP.

Accredited Training Provider

Training Provider with valid accreditation issued by the iSAQB®.

ACL

Access Control Lists control authorization of a principal to access a specific entity. An ACL attached to an entity lists principals along with their access permissions. Many file systems - among them Windows and POSIX file systems - support ACLs to control access.

Since ACLs don't scale well on a large base it is common to model access control based on roles (RBAC).

Acyclic Dependencies Principle

A fundamental principle for designing the structure of software systems (also see Package Principles). It demands that there be no cycles in the dependence graph of a system, which is also a necessity for hierarchical decomposition.

Avoiding dependence cycles is essential for low coupling and maintainability, as *all* components in a dependence cycle effectively (even if indirectly) depend on each other, which makes it hard to understand, change or replace any part of the cycle in isolation (also see [Lilienthal 2019]).

Although Robert C. Martin ([Martin 2003]) expressed it in terms of large components of object-oriented software, the ADP is a *universal* principle. It goes back (at least) to one of the origins of software architecture, the classic 1972 paper "On the Criteria To Be Used in Decomposing Systems into Modules" ([Parnas 1972]), which concludes that hierarchical structure along with "clean" decomposition are desirable properties of any system.

It can be argued that a dependence cycle, even before considering its various practical problems, is logically already as flawed as a circular argument or circular definition. As a structural contradiction, a cycle can neither be an *appropriate* nor meaningful model of the inherent nature and purpose of a system. And this conceptual divergence alone virtually guarantees for (unpredictable) problems to arise, which is exactly what a *principled* approach guards against.

Adaptability Quality Attribute

Degree to which a product or system can effectively and efficiently be adapted for different or evolving hardware, software or other operational or usage environments. Is a sub-characteristic of: portability.

Refer to [ISO-25010].

Adapter

The adapter is a design pattern that allows the interface of an existing component to be used from another interface. It is often used to make existing components cooperate with others without modifying their source code.

Aggregate

Aggregate is a building block of Domain-Driven Design. Aggregates are complex object structures that are made of entities and value objects. Each aggregate has a root entity and is regarded as one unit when it comes to changes. An aggregate ensure consistency and integrity of its contained entities with invariants.

Aggregation

A form of object composition in object-oriented programming. It differs from composition, as aggregation does not imply ownership. When the element is destroyed, the contained elements remain intact.

Analysability Quality Attribute

Degree of effectiveness and efficiency with which it is possible to assess the impact on a product or system of an intended change to one or more of its parts, or to diagnose a product for deficiencies or causes of failures, or to identify parts to be modified. Is a sub-characteristic of: maintainability.

Refer to [ISO-25010].

Appropriateness

(syn: adequacy) Suitability for a particular purpose.

Appropriateness Recognizability Quality Attribute

Degree to which users can recognize whether a product or system is appropriate for their needs. Is a sub-characteristic of: usability.

Refer to [ISO-25010].

arc42

Free and open-source template for communication and documentation of software architectures. arc42 consists of 12 (optional) parts or sections.

Architectural (Architecture) Pattern

“An architectural pattern expresses a fundamental structural organization schema for software systems. It provides a set of predefined sub-systems, specifies their responsibilities, and includes rules and guidelines for organizing the relationships between them” ([Buschmann+1996], page 12). Similar to architecture style.

Examples include:

- Layers
- Pipes and filters
- Microservices
- CQRS

Architectural Decision

Decision, which has an sustainable or essential effect on the architecture.

Example: Decision about database technology or technical basics of the user interface.

Following ISO/IEC/IEEE 42010 an architectural decision pertain to system concerns. However, there is often no simple mapping between the two. A decision can affect the architecture in several ways. These can be reflected in the architecture description (as defined in ISO/IEC/IEEE 42010).

Architectural Tactic

A technique, strategy, approach or decision helping to achieve one or several quality requirements. The term was coined by [Bass+2021].

Architecture

See Software Architecture

Architecture Description

Work product used to express an architecture (as defined in ISO/IEC/IEEE 42010).

Architecture Description Element

An architecture description element is any construct in an architecture description. architecture

description elements are the most primitive constructs discussed in ISO/IEC/IEEE 42010. All terms defined in ISO/IEC/IEEE 42010 are a specialization of the concept of an architecture description element (as defined in ISO/IEC/IEEE 42010).

Architecture Description Language

An architecture description language (ADL) is any form of expression for use in architecture descriptions (as defined in ISO/IEC/IEEE 42010).

Examples are Rapide, Wright, SysML, ArchiMate and the viewpoint languages of RM-ODP [ISO 10746].

Architecture Evaluation

Quantitative or qualitative assessment of a (software or system) architecture. Determines if an architecture can achieve its target qualities or quality attributes.

See Assessment

Comment (Gernot Starke) In my opinion the terms *architecture analysis* or *architecture assessment* are more appropriate, as *evaluation* contains *value*, implying numerical assessment or metrics, which is usually only *part* of what you should do in architecture analysis.

Architecture Framework

Conventions, principles and practices for the description of architectures established within a specific domain of application and/or community of stakeholders (as defined in ISO/IEC/IEEE 42010).

Examples are:

- Generalised Enterprise Reference Architecture and Methodologies (GERAM) [ISO 15704] is an architecture framework.
- Reference Model of Open Distributed Processing (RM-ODP) [ISO/IEC 10746] is an architecture framework.

Architecture Goal

(syn: Architectural quality goal, architectural quality requirement): A quality attribute that the system needs to achieve and the quality attribute is understood to be an architectural issue.

Hence, the architecture needs to be designed in a way to fulfill this architectural goal. These goals often have *long term character* in contrast to (short term) project goals.

Architecture Model

An architecture view is composed of one or more architecture models. An architecture model uses modelling conventions appropriate to the concerns to be addressed. These conventions are specified by the model kind governing that model. Within an architecture description, an architecture model can be a part of more than one architecture view (as defined in ISO/IEC/IEEE 42010).

Architecture Objective

See architecture goal.

Architecture Quality Requirement

See architecture goal.

Architecture Rationale

Architecture rationale records explanation, justification or reasoning about architecture decisions that have been made. The rationale for a decision can include the basis for a decision, alternatives and trade-offs considered, potential consequences of the decision and citations to sources of additional information (as defined in ISO/IEC/IEEE 42010).

Architecture Style

Description of element and relation types, together with constraints on how they can be used. Often called architecture pattern. Examples: Pipes-and-Filter, Model-View-Controller, Layers.

Comment (Alexander Lorz)

Depending on who you ask, some might consider architecture styles a generalization of architecture patterns. That is, "distributed system" is a style while "client-server, CQRS, broker and peer-to-peer" are more specific patterns that belong to this style. However, from a practical point of view this distinction is not essential.

Architecture View

A representation of a system from a specific perspective. Important and well-known views are:

- Context view
- Building block view
- Runtime view
- Deployment view

[Bass+2021] and [Rozanski+2011] extensively discuss this concept.

Following ISO/IEC/IEEE 42010, an architecture view is a work product expressing the architecture of a system from the perspective of specific system concerns (as defined in ISO/IEC/IEEE 42010).

Architecture Viewpoint

Work product establishing the conventions for the construction, interpretation and use of architecture views to frame specific system concerns (as defined in ISO/IEC/IEEE 42010).

Artifact

Tangible by-product created or generated during development of software. Examples of artifacts are use cases, all kinds of diagrams, UML models, requirements and design documents, source code, test cases, class-files, archives.

Assessment

See also Evaluation.

Gathering information about status, risks or vulnerabilities of a system. Assessment might potentially concern all aspects (development, organization, architecture, code, etc.).

Asset

"In information security, computer security and network security an Asset is any data, device, or other component of the environment that supports information-related activities. Assets generally include hardware (e.g. servers and switches), software (e.g. mission critical applications and support systems) and confidential information"

(quoted from Wikipedia)

Association

Defines a relationship between objects (in general: between components). Each association can be described in detail by cardinalities and (role-)names.

See coupling, dependency and relationship

Asymmetric Cryptography

Asymmetric cryptography algorithms are designed that the key which is used for encryption is different from the key used for decryption. The key for encryption is called "public-key" the key for decryption is called "private-key". The public key can be published and used by anyone to encrypt information only readable by the party owning the private-key for decryption. See page 17.

Asymmetric cryptography is fundamental for PKI and digital signatures.

ATAM

Architecture Tradeoff Analysis Method. Qualitative architecture evaluation method, based upon a (hierarchical) quality tree and concrete quality scenarios. Basic idea: Compare fine-grained quality scenarios ("quality-requirements") with the corresponding architectural approaches to identify risks and trade-offs.

Attack Tree

Formal way to describe different approaches of an attacker to reach certain goals. The tree is usually structured with the attack goal on top and different approaches as child nodes. Each approach is likely to have dependencies which are again listed as child nodes. The possibility of a certain way to attack an IT-system can be analyzed by assigning additional attributes to each node. Examples could be the estimated costs of an attack or if an attack approach is possible or not by referencing countermeasures.

See Bruce Schneier on "Modeling security threats".

Audit Working Group:

The *audit working group* is responsible for the technical evaluation of training materials as well as for the monitoring and evaluation of training courses. The members of the audit working group, authorized by the iSAQB®, are independent of the training provider. The result of the assessments (the respective accreditation recommendation of the audit working group) will be communicated to the training provider by the accreditation body.

Authentication

Authentication is the process of confirming the claim of an identity by a given entity. Usually this is done by verifying at least one of the authentication factors which is known by the system:

- knowledge (e.g. password)

- ownership (e.g. security token)
- inherence (e.g. biometrics)

For a stronger authentication multiple factors can be requested or at least factors of two categories.

Authenticity Quality Attribute

Degree to which the identity of a subject or resource can be proved to be the one claimed. Is a sub-characteristic of: security.

Refer to [ISO-25010].

Authorization

"Authorization or authorisation is the function of specifying access rights to resources related to information security and computer security in general and to access control in particular. More formally, "to authorize" is to define an access policy."

(quoted from Wikipedia)

Availability

One of the basic Security Goals describing a system that can provide desired information when its needed. From a security perspective for example denial-of-service-attacks may prevent availability.

Availability Quality Attribute

Degree to which a system, product or component is operational and accessible when required for use. Is a sub-characteristic of: reliability. Refer to [ISO-25010].

B

Black Box

View on a building block (or component) that hides the internal structure. Blackboxes respect the *information hiding principle*. They shall have clearly defined input- and output interfaces plus a precisely formulated *responsibility* or *objective*. Optionally a blackbox defines some quality attributes, for example timing behavior, throughput or security aspects.

Blackboard

Architectural pattern that is useful for problems for which no deterministic solution strategies are known. In blackboard, several specialized subsystems assemble their knowledge to build a possibly partial or approximate solution. (Quoted from [Buschmann+1996])

Bottom-Up Approach

Direction of work (or strategy of processing) for modeling and design. Starting with something detailed or concrete, working towards something more general or abstract.

"In a bottom-up approach the individual base elements of the system are first specified in great detail. These elements are then linked together to form larger subsystems." (quote from Wikipedia)

Bounded Context

Bounded Context is principle of the strategy design of Domain-Driven Design. "A bounded context explicitly

defines the context within which a domain model for a software system applies. Ideally, it would be preferable to have a single, unified model for all software systems in the same domain. While this is a noble goal, in reality it typically fragments into multiple models. It is useful to recognize this fact of life and work with it." (quote from Wikipedia)

"Multiple domain models are in play on any large project. Yet when code based on distinct models is combined, software becomes buggy, unreliable, and difficult to understand. Communication among team members becomes confusing. It is often unclear in what context a model should not be applied. Therefore: Explicitly set boundaries in terms of team organization, usage within specific parts of the application, and physical manifestations such as code bases and database schemas. Keep the model strictly consistent within these bounds, but don't be distracted or confused by issues outside." (quote from Wikipedia)

Bridge

Design pattern in which an abstraction is decoupled from its implementation, so that the two can vary independently. In case you find that incomprehensible (as most people)—have a look here

Broker

An architecture pattern used to structure distributed software systems with decoupled components that interact by (usually remote) service invocations.

A broker is responsible for coordinating communication, such as forwarding requests, as well as for transmitting results and exceptions.

Building Block

General or abstract term for all kinds of artifacts from which software is constructed. Part of the statical structure (Building Block View) of software architecture.

Building blocks can be hierarchically structured, they may contain other (smaller) building blocks.

Some examples of alternative (concrete) names for building blocks:
Component, module, package, namespace, class, file, program, subsystem, function, configuration, data-definition.

Building Block View

Shows the statical structure of the system, how its source code is organized. Usually a hierarchical manner, starting from the [context view]. Complemented by one or several [runtime views].

Business Architecture

A blueprint of the enterprise that provides a common understanding of the organization and is used to align strategic objectives and tactical demands.

Business Context

Shows the complete system as one blackbox within its environment from a business perspective and includes a specification of all communication partners (users, IT-systems, etc.) with explanations of domain specific inputs and outputs or interfaces. Note that the specific technical solutions for interacting with external actors should usually be omitted from the business context, as they are subject to the technical context).

See Context View.

C

C4 Model

The C4 Model for Software Architecture Documentation was developed by Simon Brown. It consists of a hierarchical set of software architecture diagrams for context, containers, components, and code. The hierarchy of the C4 diagrams provides different levels of abstraction, each of which is relevant to a different audience.

CA

A **C**ertificate **A**uthority issues digital certificates to a given subject in a PKI. Usually there is a trust established to this authority which results in the same trust level to the issued certificates.

An example is the widely used TLS-PKI where every browser includes the root-certificates of a defined list of CAs. These CAs then check the identity of a given internet domain owner and digitally sign his certificate for the use with TLS.

Capacity Quality Attribute

Degree to which the maximum limits of a product or system parameter meet requirements. Is a sub-characteristic of: performance efficiency.

Refer to [ISO-25010].

Cardinality

Describes the quantitative rating of an association or relationship. It specifies the number of participants (objects, instances, modules etc) of the association.

Certification Program

The iSAQB® CPSA® certification program, including its organizational components, documents (training documents, contracts) and processes.

The copyrighted abbreviation and term CPSA® means **Certified Professional for Software Architecture**.

CIA Triad

See Security Goals

Clean Architecture

See Ports and Adapters.

Closure of Operation

See Combinator.

Cloud

"Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction."

Definition quoted from NIST (National Institute of Standards and Technology).

The NIST definition contains the following five characteristics (quoted but abbreviated from the aforementioned NIST source too):

- On-demand self-service: A consumer can unilaterally provision computing capabilities, such as server time and network storage, without requiring human interaction with each service provider.
- Broad network access: Capabilities are available over the network and accessed through standard mechanisms that promote use by heterogeneous client platforms.
- Resource pooling: The provider's computing resources are pooled to serve multiple consumers using a multi-tenant model, with different physical and virtual resources dynamically assigned and reassigned according to consumer demand. There is location independence in that the customer generally has no control or knowledge over the exact location of the provided resources but may be able to specify location at a higher level of abstraction (e.g., country, state, or datacenter). Examples of resources include storage, processing, memory, and network bandwidth.
- Rapid elasticity: Capabilities can be elastically provisioned and released, in some cases automatically, to scale rapidly commensurate with demand. To the consumer, the capabilities available for provisioning often appear to be unlimited and can be appropriated in any quantity at any time.
- Measured service: Cloud systems automatically control and optimize resource use by leveraging a metering capability at some level of abstraction appropriate to the type of service (e.g., storage, processing, bandwidth, and active user accounts). Resource usage can be monitored, controlled, and reported, providing transparency for both the provider and consumer of the utilized service.

Co-Existence Quality Attribute

Degree to which a product can perform its required functions efficiently while sharing a common environment and resources with other products, without detrimental impact on any other product. Is a sub-characteristic of: compatibility.

Refer to [ISO-25010].

Cohesion

The degree to which elements of a building block, component or module belong together is called cohesion. It measures the strength of relationship between pieces of functionality within a given component. In cohesive systems, functionality is strongly related. It is usually characterized as *high cohesion* or *low cohesion*. Strive for high cohesion, because high cohesion often implies reusability, low coupling and understandability.

Combinator

Design pattern to build complex functions or objects by combining simpler ones. For some domain object of type T, look for operations with both input and output type T. Also known as *closure of operation*. See [Yorgey 2012] and [Maguire 2019].

Command

Design pattern in which an object is used to encapsulate an action. This action might be invoked or executed at a later time.

Common Closure Principle

A fundamental principle for designing the structure of software systems (also see Package Principles). It directly and explicitly restates the Single Responsibility Principle for larger components.

The subcomponents of a component should ideally have the exact same reasons to change. A change request that effects one of them should effect all of them, but it should **not** affect anything else outside their enclosing component.

Thereby, each expected change request would effect a minimal number of components. Or put another way: Each component would be closed to a maximum number of expected change requests. The term **expected** here signifies a few important implications:

1. The inherent concepts/responsibilities of a system run deeper than a surface-level description of its behaviour.
2. The deeper concepts/responsibilities of a system are not entirely objective but can be modeled in different ways.
3. Determining the concepts/responsibilities of a system is not just passive describing but also active **strategizing**.

This principle leads to highly cohesive components. It also implies loosely coupled components because related concepts that **do** change together **do** get bundled up in the same component. When each single concept is expressed by a single component, there are no unnecessary couplings between components.

Common Reuse Principle

A fundamental principle for designing the structure of software systems (also see Package Principles). The subcomponents (classes) of a component should be exactly the ones that are being (re)used together. Or the other way around: Components that are being (re)used together should be packaged into a larger component. This also implies that subcomponents that are **not** frequently used in conjunction with the other subcomponents should **not** be in the respective component.

This perspective helps in deciding what belongs into a component and what doesn't. It aims at a system decomposition of loosely coupled and highly cohesive components.

This obviously echoes the Single Responsibility Principle. It also echoes the Interface Segregation Principle, as it ensures that clients aren't forced to depend on concepts they don't care about.

Compatibility Quality Attribute

Degree to which a product, system or component can exchange information with other products, systems or components, and/or perform its required functions, while sharing the same hardware or software environment. Is composed of the following sub-characteristics: co-existence, interoperabilty.

Refer to [ISO-25010].

Complexity

"Complexity is generally used to characterize something with many parts where those parts interact with each other in multiple ways." (quoted from Wikipedia.)

- *Essential* complexity is the core of the problem we have to solve, and it consists of the parts of the software that are legitimately difficult problems. Most software problems contain some complexity.
- *Accidental* complexity is all the stuff that doesn't necessarily relate directly to the solution, but that we have to deal with anyway.

(quoted from Mark Needham)

Architects shall strive to reduce accidental complexity.

Component

See Building block. Structural element of an architecture.

Composition

Combine simpler elements (e.g. functions, data types, building blocks) to build more complicated, powerful or more responsible ones.

In UML: When the owning element is destroyed, so are the contained elements.

Concept

Plan, principle(s) or rule(s) how to solve a specific problem.

Concepts are often *cross-cutting* in a sense that multiple architectural elements might be affected by a single concept. That means that implementors of e.g. implementation units (building blocks) should adhere to the corresponding concept.

Concepts form the basis for conceptual integrity.

Conceptual Integrity

Concepts, rules, patterns and similar solution approaches are applied in a consistent (homogeneous, similar) way throughout the system. Similar problems are solved in similar or identical ways.

Concern

"A *concern* about an architecture is a requirement, an objective, a constraint, an intention, or an aspiration a stakeholder has for that architecture." (quoted from [Rozanski+2011], chapter 8)

Following ISO/IEC/IEEE 42010 a concern is defined as (system) interest in a system relevant to one or more of its stakeholders (as defined in ISO/IEC/IEEE 42010).

Note, a concern pertains to any influence on a system in its environment, including developmental, technological, business, operational, organizational, political, economic, legal, regulatory, ecological and social influences.

Concurrency

Concurrency is the ability of different parts or units of a program, algorithm, or problem to be executed out-of-order or in partial order, without affecting the final outcome. Concurrency does not necessarily mean parallelism. (nach Wikipedia)

Confidentiality

One of the basic Security Goals describing a system to disclose and make information only available to authorized parties.

Confidentiality Quality Attribute

Degree to which a product or system ensures that data are accessible only to those authorized to have access. Is a sub-characteristic of: security.

Refer to [ISO-25010].

Consistency

A consistent systems does not contain contradictions.

- Identical problems are solved with identical (or at least similar) approaches.
- Degree, to which data and their relations comply to validation rules.
- Clients (of a database) get identical results for identical queries (e.g. Monotonic-Read-Consistency, Monotonic-Write-Consistency, Read-Your-Writes-Consistency etc.)
- With respect to behavior: Degree, to which a system behaves coherent, replicable and reasonable.

Constraint

A restriction on the degree of freedom you have in creating, designing, implementing or otherwise providing a solution. Constraints are often *global requirements*, such as limited development resources or a decision by senior management that restricts the way you plan, design, develop or operate a system.

Based upon a definition from Scott Ambler

Context (of a System)

"Defines the relationships, dependencies, and interactions between the system and its environment: People, systems, and external entities with which it interacts." (quoted from Rozanski-Woods)

Another definition from arc42: "System scope and context - as the name suggests - delimits your system (i.e. your scope) from all its communication partners (neighboring systems and users, i.e. the context of your system). It thereby specifies the external interfaces." (quoted from docs.arc42.org)

Distinguish between *business* and *technical* context:

- The **business** context (formerly called *logical* context) shows the external relationships from a business- or non-technical perspective. It abstracts from technical, hardware or implementation details. Input-/Output relationships are named by their *business meaning* instead of their technical properties.
- The **technical** context shows technical details, like transmission channel, technical protocol, IP-address, bus or similar hardware details. Embedded systems, for example, often care for hardware-related information very early in development.

Context View

Shows the complete system as one blackbox within its environment. This can be done from a business perspective (*business context*) and/or from a technical or deployment perspective (*technical context*). The context view (or context diagram) shows the boundary between a system and its environment, showing the entities in its environment (its neighbors) with which it interacts.

Neighbors can either be other software, hardware (like sensors), humans, user-roles or even organizations using the system.

See Context.

Correspondence

A correspondence defines a relation between architectural description elements. Correspondences are used to express architecture relations of interest within an architecture description (or between architecture descriptions) (as defined in ISO/IEC/IEEE 42010).

Correspondence Rule

Correspondences can be governed by correspondence rules. Correspondence rules are used to enforce relations within an architecture description (or between architecture descriptions) (as defined in ISO/IEC/IEEE 42010).

Synonym: Integrity, homogeneity, conceptual integrity.

Coupling

Coupling is the kind and degree of *interdependence* between building blocks of software; a measure of how closely connected two components are.

You should always aim for *low* coupling. Coupling is usually contrasted with cohesion. Low coupling often correlates with high cohesion, and vice versa. Low coupling is often a sign of a well-structured system. When combined with high cohesion, it supports understandability and maintainability.

CPSA®

Certified Professional for Software Architecture® – the common name for different levels of certification issued by the iSAQB. The most common known certifications are the foundation level (CPSA-F) and the advanced level (CPSA-A).

CQRS

(command query responsibility segregation): Separate the elements manipulating (*command*) data from those just reading (*query*). This separation enables different optimization strategies for reading and writing data (for example, it's much easier to cache data that's read-only than to cache data that's also altered.)

There's an interesting eBook by Mark Nijhof on this subject.

Cross-Cutting Concept

See concept.

Synonym: principle, rule.

Cross-Cutting Concern

Functionality of the architecture or system that affects several elements. Examples of such concerns are logging, transactions, security, exception handling, caching etc.

Often these concerns will be addressed in systems via concepts.

Curriculum

The learning process provided by a school (here: iSAQB® as the institution governing software architecture education). It includes the content of courses (the syllabus), the methods employed, and other aspects, like norms and values, which relate to the way the education including certification and examination is organized.

Cyclomatic Complexity

Quantitative measure, number of independent paths through a program's source code. It roughly correlates to the number of conditional statements (`if`, `while`) in the code +1. A linear sequence of statements without `if` or `while` has the cyclomatic complexity of 1. Many software engineers believe that higher complexity correlates to the number of defects.

D

Decomposition

(syn: factoring) Breaking or dividing a complex system or problem into several smaller parts that are easier to understand, implement or maintain.

Dependency

See coupling.

Dependency Injection (DI)

Instead of having your objects or a factory creating a dependency, you pass the needed dependencies to the constructor or via property setters. You therefore make the creation of specific dependencies *somebody else's problem*. Often used to ensure the dependency inversion principle.

Dependency Inversion Principle

High level (abstract) elements should not depend upon low level (specific) elements. "Details should depend upon abstractions" ([Martin 2003]). One of the SOLID principles, nicely explained by Brett Schuchert, and closely related to the SDP and SAP.

Deployment

Bring software onto its execution environment (hardware, processor etc). Put software into operation.

Deployment View

Architectural view showing the technical infrastructure where a system or artifacts will be deployed and executed.

"This view defines the physical environment in which the system is intended to run, including the hardware environment your system needs (e.g., processing nodes, network interconnections, and disk storage facilities), the technical environment requirements for each node (or node type) in the system, and the mapping of your software elements to the runtime environment that will execute them." (as defined by Rozanski+2011)

Design Pattern

General or generic reusable solution to a commonly occurring problem within a given context in design. Initially conceived by the famous architect Christopher Alexander, the concept of *design patterns* was taken up by software engineers.

In our opinion, every serious software developer should know at least some patterns from the pioneering Gang-of-Four book by Erich Gamma ([GoF: Design-Patterns]) and his three allies.

Design Principle

Set of guidelines that helps software developers to design and implement better solutions, where "better" could, for example, mean one or more of the following:

- low coupling.
- high cohesion.
- separation of concerns or adherence to the Single Responsibility Principle.
- adherence to the Information Hiding principle.
- avoid **Rigidity**: A system or element is difficult to change because every change potentially affects many other elements.
- avoid **Fragility**: When elements are changed, unexpected results, defects or otherwise negative consequences occur at other elements.
- avoid **Immobility**: An element is difficult to reuse because it cannot be disentangled from the rest of the system.

Design Rationale

An explicit documentation of the reasons behind decisions made when designing any architectural element.

Document

A (usually written) artifact conveying information.

Documentation

A systematically ordered collection of documents and other material of any kind that makes usage or evaluation easier. Examples for "other material": presentation, video, audio, web page, image, etc.

Documentation Build

Automatic build process that collects artifacts into a consistent documentation.

Domain Model

The domain model is a concept of Domain-Driven Design. It provides a system of abstractions that describes selected aspects of a domain and can be used to solve problems related to that domain.

Domain-Driven Design (DDD)

"Domain-driven design (DDD) is an approach to developing software for complex needs by deeply connecting the implementation to an evolving model of the core business concepts."

(quoted from DDDCommunity). See [Evans 2004].

See also:

- Entity
- Value Object
- Aggregate
- Service
- Factory
- Repository

- Ubiquitous Language

Drawing Tool

A tool to create drawings that can be used in architecture documentation. Example: MS Visio, OmniGraffle, PowerPoint, etc. Drawing tools treat each drawing as a separate thing, this causes maintenance overhead when updating an element of the architecture that is shown in several diagrams (as opposed to a Modeling Tool).

E

Economicalness

Being economical, simple, lean or achievable with relatively low effort.

Embedded System

System *embedded* within a larger mechanical or electrical system. Embedded systems often have real-time computing constraints. Typical properties of embedded systems are low power consumption, limited memory and processing resources, small size.

Encapsulation

Encapsulation has two slightly distinct notions, and sometimes the combination thereof:

- restricting access to some of the object's components
- bundling of data with the methods or functions operating on that data

Encapsulation is a mechanism for information hiding.

Enterprise IT Architecture

Synonym: Enterprise Architecture.

Structures and concepts for the IT support of an entire company. Atomic subject matters of the enterprise architecture are single software systems also referred to as "applications".

Entity

Entity is a building block of Domain-Driven Design. An entity is a core object of a business domain with unchangeable identity and a clearly defined lifecycle. Entities map their state to value objects and are almost always persistent.

Entropy

In information theory defined as "amount of information" a message has or "unpredictability of information content". The entropy of a crypto system is measured by the size of the keyspace. Larger keyspaces have an increased entropy and, if not flawed by the algorithm itself, are harder to break than smaller ones. For secure cryptographic operations it is mandatory to not only use random values as input, they should have also a high entropy. The creation of high entropy on a computer system is non-trivial and can affect the performance of a system.

See [Schneier 1996] and Whitewood Inc. on "Understanding and Managing Entropy" or SANS "Randomness and Entropy - An Introduction".

Environment

(System) Context determining the setting and circumstances of all influences upon a system (as defined in ISO/IEC/IEEE 42010).

Note, the environment of a system includes developmental, technological, business, operational, organizational, political, economic, legal, regulatory, ecological and social influences.

Event Sourcing

Architecture pattern where changes to the application's state are captured as a series of immutable events. Instead of storing just the current state of the application, every state change is recorded as an event in an append-only log.

F

Facade

Structural design pattern. A Facade offers a simplified interface to a complex or complicated building block (the *provider*) without any modifications to the provider.

Factory

(Design pattern) In class-based or object-oriented programming, the factory method pattern is a creational design pattern that uses factory methods or factory components for creating objects, without having to specify the exact class of the object that will be created.

In Domain-Driven Design: A factory encapsulates the creation of aggregates, entities, and value objects. Factories work exclusively in the domain and have no access to technical building blocks (e.g. a database).

Fault Tolerance Quality Attribute

Degree to which a system, product or component operates as intended despite the presence of hardware or software faults. Is a sub-characteristic of: reliability.

Refer to [ISO-25010].

Filter

Part of the pipes and filters architectural pattern used to create or transform data. Filters are typically executed independently of other filters. See pipes and filters.

Fitness Function

"An architectural fitness function provides an objective integrity assessment of some architectural characteristics." ([Ford+2017]).

A fitness function is derived from manual evaluations and automated tests and shows to which extent architectural or quality requirements are met.

Function Signature

(Synonym: type or method signature) defines input and output of functions or methods.

A signature can include:

- parameters and their types
- return value and type
- exception thrown or errors raised

Functional Appropriateness Quality Attribute

Degree to which the functions facilitate the accomplishment of specified tasks and objectives. Is a sub-characteristic of: functional suitability.

Refer to [ISO-25010].

Functional Completeness Quality Attribute

Degree to which the set of functions covers all the specified tasks and user objectives. Is a sub-characteristic of: functional suitability.

Refer to [ISO-25010].

Functional Correctness Quality Attribute

Degree to which a product or system provides the correct results with the needed degree of precision. Is a sub-characteristic of: functional suitability.

Refer to [ISO-25010].

Functional Suitability Quality Attribute

Degree to which a product or system provides functions that meet stated and implied needs when used under specified conditions. Is composed of the following sub-characteristics: functional completeness, functional correctness, functional appropriateness.

Refer to [ISO-25010].

Fundamental Modeling Concepts (FMC)

Fundamental Modeling Concepts is a graphical notation for modeling and documenting software systems. From their website:
"FMC provide a framework for the comprehensive description of software-intensive systems. It is based on a precise terminology and supported by a graphical notation which can be easily understood".

G

Gateway

A (design or architecture) pattern: An element of that encapsulates access to a (usually external) system or resource. See also wrapper, adapter.

Global Analysis

Systematic approach to achieve desired quality attributes. Developed and documented by Christine Hofmeister (Siemens Corporate Research). Global analysis is described in [Hofmeister+2000].

H

Heterogeneous Architectural Style

See hybrid architecture style.

Heuristic

Informal rule, rule-of-thumb. Any way of problem-solving not guaranteed to be optimal, but somehow sufficient. Examples from Object-Oriented Design or User Interface Design.

Hexagonal Architecture

See Ports and Adapters.

Hybrid Architecture Style

Combination of two or more existing architecture styles or patterns. For example, an MVC construct embedded in a layer structure.

I

IEEE-1471

Standard *Recommended Practice for Architectural Description of Software-Intensive Systems*, defined as ISO/IEC 42010:2007. Defines a (abstract) framework for the description of software architectures.

Incremental Development

See iterative and incremental development.

Information Hiding

A fundamental principle in software design: Keep those design or implementation decisions *hidden* that are likely to change, thus protecting other parts of the system from modification if these decisions or implementations are changed. Is one important attributes of blackboxes. Separates interface from implementation.

The term encapsulation is often used interchangeably with information hiding.

Installability Quality Attribute

Degree of effectiveness and efficiency with which a product or system can be successfully installed and/or uninstalled in a specified environment. Is a sub-characteristic of: portability.

Refer to [ISO-25010].

Integrity

Various meanings:

One of the basic security goals which means maintaining and assuring accuracy and completeness of data. Usually this is achieved by the usage of cryptographic algorithms to create a digital signature.

Data or behavioral integrity:

- Degree to which clients (of a database) get identical results for identical queries (e.g. Monotonic-Read-Consistency, Monotonic-Write-Consistency, Read-Your-Writes-Consistency etc.)
- Degree, to which a system behaves coherent, replicable and reasonable.

See also consistency.

Integrity Quality Attribute

Degree to which a system, product or component prevents unauthorized access to, or modification of, computer programs or data. Is a sub-characteristic of: security.

Refer to [ISO-25010].

Interface

Multiple meanings, depending on context:

1. Boundary across which two building blocks interact or communicate with each other.
2. Design construct that provides an abstraction of the behavior of concrete components, declares possible interactions with these components and constraints for these interactions.

An example for the second meaning is the programming language construct interface from the object-oriented language Java(tm):

```
/* File name : Animal.java */
interface Animal {
   public void eat();
   public void move();
}

/* File name : Horse.java */
public class Horse implements Animal {

   public void eat() {
      System.out.println("Horse eats");
   }

   public void move() {
      System.out.println("Horse moves");
   }
}
```

Interface Segregation Principle (ISP)

Building blocks (classes, components) should not be forced to depend on methods they don't use. ISP splits larger interfaces into smaller and more (client) specific ones so that clients will only need to know about methods that they actually use.

Interoperability Quality Attribute

Degree to which two or more systems, products or components can exchange information and use the information that has been exchanged. Is a sub-characteristic of: compatibility.

Refer to [ISO-25010].

Interpreter

Design pattern that represents domain objects or domain-specific languages as syntax. An interpreter function then provides a semantic interpretation of domain objects separately from the objects.

iSAQB®

international Software Architecture Qualification Board – an internationally active organization fostering the development of software architecture education. See also the discussion in the appendix.

ISO 25010

Official name:

Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE)

It is a standard to describe *software product quality*.

ISO proposes a hierarchical model of product quality, with currently (standard version 2011) 8 top-level attributes:

image::1_architecture/ISO-25010-EN.png

The future version of the ISO standard will contain 9 of these attributes.

For a list of quality attributes defined by the ISO 25010 standard, refer to [ISO-25010].

Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE)

ISO 9126

(Deprecated) standard to describe (and evaluate) *software product quality*. Has been superseded by ISO 25010.

Iterative and Incremental Development

Combination of iterative and incremental approaches for software development. These are essential parts of the various *agile* development approaches, e.g. Scrum and XP.

Iterative Development

"Development approach that *cycles* through development phases, from gathering requirements to delivering functionality in a working release." (quoted from c2-wiki)

Such cycles are repeated to improve either functionality, quality or both.

Contrast to the Waterfall Development.

K

Kerckhoffs' Principle

One of the six cryptographic axioms described 1883 in an article "La cryptographie militaire" by the Dutch cryptographer and linguist Auguste Kerckhoffs. This axiom is still relevant today and therefore referred to as "Kerckhoffs' Principle".

It describes that a cryptographic method must not need to be kept secret in order to achieve the security of the encrypted messages.

"The enemy knows the system" is another expression coined by the mathematician Claude Shannon as Shannon's Maxim.

See Bruce Schneiers Crypto-Gram, May 15, 2002

L

Latency

Latency is the time delay between the cause and the effect of some change in a system.

In computer networks, latency describes the time it takes for an amount of data (*packet*) to get from one specific location to another.

In interactive systems, latency is the time interval between some input to the system and the audio-visual response. Often a delay exists, often caused by network delays.

Layer

Grouping of building blocks or components that (together) offer a cohesive set of services to other layers. Layers are related to each other by the ordered relation *allowed to use*. Two common kinds of layers are *abstraction layers* to hide details (example: ISO/OSI network layers, or "hardware abstraction layer", see https://en.wikipedia.org/wiki/Hardware_abstraction) and layers that (physically) separate functionality or responsibility (see https://en.wikipedia.org/wiki/Multitier_architecture).

Learnability Quality Attribute

Degree to which a product or system can be used by specified users to achieve specified goals of learning to use the product or system with effectiveness, efficiency, freedom from risk and satisfaction in a specified context of use. Is a sub-characteristic of: usability.

Refer to [ISO-25010].

Liskov Substitution Principle

Refers to object-oriented programming: If you use inheritance, do it right: Instances of derived types (subclasses) must be completely substitutable for their base types. If code uses a base class, these references can be replaced with any instance of a derived class without affecting the functionality of that code.

M

Maintainability Quality Attribute

Degree of effectiveness and efficiency with which a product or system can be modified to improve it, correct it or adapt it to changes in environment, and in requirements. Is composed of the following sub-characteristics: modularity, reusability, analysability, modifiability, testability.

Refer to [ISO-25010].

Maturity Quality Attribute

Degree to which a system, product or component meets needs for reliability under normal operation. Is a sub-characteristic of: reliability.

Refer to [ISO-25010].

MFA

For Multi-Factor-Authentication see Authentication.

Microservice

An architectural style, proposing to divide large systems into small units. "Microservices have to be implemented as virtual machines, as more light-weight alternatives such as Docker containers or as individual processes. Thereby they can easily be brought into production individually." (quoted from the (free) LeanPub booklet on Microservices by Eberhard Wolff)

Model Driven Architecture (MDA)

Model Driven Architecture (MDA) is an OMG-Standard for model based software development. Definition: "An approach to IT system specification that separates the specification of functionality from the specification of the implementation of that functionality on a specific technology platform."

Model Kind

Conventions for a type of modeling (as defined in ISO/IEC/IEEE 42010).

Note, examples of model kinds include data flow diagrams, class diagrams, Petri nets, balance sheets, organization charts and state transition models.

Model-Driven Software Development (MDSD)

The underlying idea is to generate code from more abstract models of requirements or the domain.

Model-View-Controller (MVC)

Architecture pattern, often used to implement user interfaces. It divides a system into three interconnected parts (model, view, and controller) to separate the following responsibilities:

- Model manages data and logic of the system. The "truth" that will be shown or displayed by one or many views. Model does not know (depend on) its views.
- View can be any number of (arbitrary) output representation of (model) information. Multiple views of the same model are possible.
- Controller accepts (user) input and converts those to commands for the model or view.

Model-View-Update (MVU)

Architecture pattern, often used to implement user interfaces. It emphasizes immutability and unidirectional data flow. It consists of three parts:

- Model represents the application’s state as an immutable data structure.
- View is a function without side effects for rendering the model in the UI.
- Update is a function that handles updates on the model by producing a new model instance.

Model-View-ViewModel (MVVM)

Architecture pattern, often used to implement user interfaces. It divides a system into three interconnected parts (model, view, and view model) to separate the following responsibilities:

- Model manages data and domain logic of the system. Does not depend on the view and the view model.
- View is the visible user interface of the application (or parts thereof).
- ViewModel serves as an intermediary between the View and the Model and holds the UI logic. May depend on the model but not on the view.

Modeling Tool

A tool that creates models (e.g. UML or BPMN models). Can be used to create consistent diagrams for documentation because it has the advantage that each model element exists only once but can be consistently displayed in many diagrams (as opposed to a mere Drawing Tool).

Modifiability Quality Attribute

Degree to which a product or system can be effectively and efficiently modified without introducing defects or degrading existing product quality. Is a sub-characteristic of: maintainability.

Refer to [ISO-25010].

Modular Programming

"Software design technique that separates the functionality of a program into independent, interchangeable *modules*, so that each module contains everything necessary to execute only one aspect of the desired functionality.

Modules have *interfaces* expressing the elements provided and required by the module. The elements defined in the interface are detectable by other modules." (quoted from Wikipedia)

Modularity Quality Attribute

Degree to which a system or computer program is composed of discrete components such that a change to one component has minimal impact on other components. Is a sub-characteristic of: maintainability.

Refer to [ISO-25010].

Module

(see also Modular programming)

1. structural element or building block, usually regarded as a *black box* with a clearly defined responsibility. It encapsulates data and code and provides public interfaces, so clients can access its functionality. This meaning has first been described in a groundbreaking and fundamental paper from David L. Parnas: On the Criteria to be Used in Decomposing Software into Modules
2. In several programming languages, *module* is a construct for aggregating smaller programming units, e.g. in Python. In other languages (like Java), modules are called *packages*.
3. The CPSA®-Advanced Level is currently divided into several modules, which can be learned or taught separately and in any order. The exact relationships between these modules and the contents of these modules are defined in the respective curricula.

N

Node (in UML)

A processing resource (execution environment, processor, machine, virtual machine, application server) where artifacts can be deployed and executed.

Node (Node.js)

In modern web development: Short form for the open-source JavaScript runtime Node.js®, which is built on Chrome's V8 JavaScript engine. Node.js is famous for its an event-driven, non-blocking I/O model and its vast ecosystem of supporting libraries.

Non Functional Requirement (NFR)

Requirements that *constrain the solution*. Nonfunctional requirements are also known as *quality attribute requirements* or quality requirements. The term NFR is actually misleading, as many of the *attributes* involved directly relate to specific system *functions* (so modern requirements engineering likes to call these things *required constraints*).

Non-repudiation Quality Attribute

Degree to which actions or events can be proven to have taken place, so that the events or actions cannot be repudiated later. Is a sub-characteristic of: security.

Refer to [ISO-25010].

Notation

A system of marks, signs, figures, or characters that is used to represent information. Examples: prose, table, bullet point list, numbered list, UML, BPMN.

O

Observer

(Design pattern) "… in which an object, called the subject, maintains a list of its dependents, called observers, and notifies them automatically of any state changes, usually by calling one of their methods." (quoted from Wikipedia)

The Observer pattern is a key pattern to complement the model-view-controller (MVC) architectural pattern.

Onion Architecture

See Ports and Adapters.

Open-Close-Principle (OCP)

"Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification" (Bertrand Meyer, 1998). In plain words: To *add* functionality (extension) to a system, you should *not need to modify* existing code. Part of Robert Martin's "SOLID" principles for object-oriented systems. Can be implemented in object-oriented languages by interface inheritance, in a more general way as *plugins*.

Operability Quality Attribute

Degree to which a product or system has attributes that make it easy to operate and control. Is a sub-characteristic of: usability.

Refer to [ISO-25010].

OWASP

The **Open Web Application Security Project** is a worldwide non-profit online organization founded 2001 for improving the security of software. It is a rich source for information and best practices in the field of web security. See https://www.owasp.org/.

The OWASP-Top-10 is a frequently referenced list of attack categories based on the projects data survey.

P

Package Principles

Fundamental principles for designing the structure of software systems ([Martin 2003]):

- Reuse/Release Equivalence Principle (REP)
- Common Reuse Principle (CRP)
- Common Closure Principle (CCP)
- Acyclic Dependencies Principle (ADP)
- Stable Dependencies Principle (SDP)
- Stable Abstractions Principle (SAP)

Robert C. Martin, who coined the "SOLID" acronym, also introduced the package principles) and frequently reiterated both in conjunction. Whereas the SOLID principles target the level of classes, the package principles target the level of larger components that contain multiple classes and might get deployed independently.

Package and SOLID principles share the explicit goal of keeping software maintainable and avoiding the symptoms of degraded design: rigidity, fragility, immobility, and viscosity.

While Martin expressed the Package Principles in terms of large-scale components, they apply at other scales as well. Their core are universal principles like low coupling, high cohesion, single responsibility, hierarchical (acyclic) decomposition, and the insight that meaningful dependencies go from specific/unstable concepts to more abstract/stable ones (which echoes the DIP).

Pattern

```
A reusable or repeatable solution to a common problem in software design or
architecture.
```

See architecture pattern or design pattern.

Perfect Forward Secrecy

Property of a cryptographic protocol were an attacker can't gain any information about short-term session keys by compromising long-term keys.

Examples for protocols with perfect forward secrecy are TLS and OTR. If this feature is enabled for TLS and an attacker gains access to a servers private key, previously recorded communication sessions can

still not be decrypted.

Performance Efficiency Quality Attribute

Performance relative to the amount of resources used under stated conditions.

Resources can include other software products, the software and hardware configuration of the system, and materials (e.g. print paper, storage media).

Is composed of the following sub-characteristics: time behaviour, resource utilization, capacity.

Refer to [ISO-25010].

Perspective

A perspective is used to consider a set of related quality properties and concerns of a system.

Architects apply perspectives iteratively to the system's *architectural views* in order to assess the effects of *architectural design decisions* across multiple *viewpoints* and *architectural views*.

[Rozanski+2011] associates with the term *perspective* also activities, tactics, and guidelines that must be considered if a system should provide a set of related quality properties and suggests the following perspectives:

- Accessibility
- Availability and Resilience
- Development Resource
- Evolution
- Internationalization
- Location
- Performance and Scalability
- Regulation
- Security
- Usability

Pikachu

A yellowish mouse-like character from the (quite famous) Pokémon world. Actually, you don't need to know that. But it does not hurt either - and you might impress your kids with this knowledge…

Pipe

Connector in the pipes-and-filters architectural style that transfers streams or chunks of data from the output of one filter to the input of another filter without modifying values or order of data. See pipes and filters.

Pipes and Filters

The Pipes and Filters architectural pattern provides a structure for systems that process a stream of data. Each processing step is encapsulated in a filter component. Data is passed through pipes between

adjacent filters. Recombining filters allows you to build families of related systems. (Quoted from [Buschmann+1996], also see pipe and filter.)

PKI

Short for **Public-Key-Infrastructure**. A concept of managing digital certificates usually involving asymmetric cryptography. The term "public" refers most of the time to the used type of cryptographic key and not necessarily to infrastructure open to a public audience. To prevent semantic confusion the terms "open PKI" or "closed PKI" can be used, see [Anderson 2008] Chapter 21.4.5.7 PKI, page 672.

PKI is usually based on a CA or a Web-of-Trust.

Plugin

Architecture pattern that allows the extension of an application's functionality without modifying its core. This is achieved by enabling external modules, known as plugins, to add features or interact with the core application.

Port

UML construct, used in component diagrams. An interface, defining a point of interaction of a component with its environment.

Portability Quality Attribute

Degree of effectiveness and efficiency with which a system, product or component can be transferred from one hardware, software or other operational or usage environment to another. Is composed of the following sub-characteristics: adaptability, installability, replaceability.

Refer to [ISO-25010].

Ports and Adapters

Architecture pattern that concentrates the domain logic in the center of the system. Ports connect the domain logic with the outside world, independent from a specific technology. Different adapters attach to ports to translate requests and responses for a certain technology. This approach allows an application to be driven by different agents (e.g. users, programs, automated tests), and to be developed and tested in isolation from its production environment. See [Cockburn 2005], [Lange 2021], [Hombergs 2024].

Also known as Onion Architecture, Hexagonal Architecture, Clean Architecture.

POSA

Pattern-oriented Software Architecture. Series of books on software architecture patterns.

Presentation-Abstraction-Control (PAC)

Architecture pattern used for structuring interactive software systems into a hierarchy of cooperating agents. Each agent is divided into three distinct components:

- Presentation is the user interface part of the agent.
- Abstraction handles domain logic and data.
- Control manages interactions between presentation and abstraction.

Principal

Principals in a security context are entities which have been authenticated and can be assigned permissions to. A principal can be a user but for example also other services or a process running on a system. The term is used in the Java environment and throughout different authentication protocols (see GSSAPI RFC2744 or Kerberos RFC4121).

Product

Product data is data whose values have several, fixed attributes. Also known as **records**, **structs**, **tuples** or **and data**.

Example:

An **address** has the following attributes:

- name, and
- street, and
- street number, and
- city, and
- zip code

Also see [Sperber+2024].

Proxy

(Design pattern) "A wrapper or agent object that is being called by the client to access the real serving object behind the scenes. Use of the proxy can simply be forwarding to the real object, or can provide additional logic. In the proxy extra functionality can be provided, for example caching when operations on the real object are resource intensive, or checking preconditions before operations on the real object are invoked. For the client, usage of a proxy object is similar to using the real object, because both implement the same interface." (quoted from Wikipedia)

Pseudo-Randomness

Often used in conjunction with pseudo-random-number-generators. Gathering randomness with a high entropy is resource intensive and usually not required by many applications, cryptography left aside. To address this issue pseudo-random-generators are initialized with a seed of data and create random values based on this seed. The data will be generated by random, but will always be the same if the generator is initialized with an identical seed. This is called pseudo-randomness and is less performance intensive.

Q

Qualitative Evaluation

Finding risks concerning the desired quality attributes of a system. Analyzing or assessing if a system or its architecture can meet the desired or required quality goals.

Instead of calculating or measuring certain characteristics of systems or architectures, qualitative evaluation is concerned with risks, trade-offs and sensitivity points.

See also assessment.

Quality

See software quality and quality attributes.

Quality Attribute

Software quality is the degree to which a system possesses the desired combination of *attributes* (see: software quality).

The Standard ISO-25010 defines the following quality attributes:

- Functional suitability
  - Functional completeness
  - Functional correctness
  - Functional appropriateness
- Performance efficiency
  - Time behaviour
  - Resource utilization
  - Capacity
- Compatibility
  - Co-existence
  - Interoperability
- Usability
  - Appropriateness recognizability
  - Learnability
  - Operability
  - User error protection
  - User interface aesthetics
  - Accessibility
- Reliability
  - Availability
  - Fault tolerance
  - Recoverability
- Security
  - Confidentiality
  - Integrity
  - Non-repudiation
  - Accountability
  - Authenticity
- Maintainability

    - Modularity
    - Reusability
    - Analysability
    - Modifiability
    - Testability
- Portability
    - Adaptability
    - Installability
    - Replaceability

It might be helpful to distinguish between the following types of quality attributes:

- *runtime quality attributes* (which can be observed at execution time of the system),
- *non-runtime quality attributes* (which cannot be observed as the system executes) and
- *business quality attributes* (cost, schedule, marketability, appropriateness for organization)

Examples of runtime quality attributes are functional suitability, performance efficiency, security, reliability, usability and interoperability.

Examples of non-runtime quality attributes are modifiability, portability, understandability and testability.

Quality Characteristic

synonym: quality attribute.

Quality Model

(from ISO 25010) A model that defines quality characteristics that relate to static properties of software and dynamic properties of the computer system and software products. The quality model provides consistent terminology for specifying, measuring and evaluating system and software product quality.

The scope of application of the quality models includes supporting specification and evaluation of software and software-intensive computer systems from different perspectives by those associated with their acquisition, requirements, development, use, evaluation, support, maintenance, quality assurance and control, and audit.

Comment (Gernot Starke)

A quality model (like ISO-25010) **only** provides a taxonomy of terms, but does **not** provide any means to specify or evaluate quality. I consent to the phrase above "consistent terminology", but strongly object to "measuring and evaluating". For measuring and evaluating you definitely need additional tools and/or methods, the pure model does not help.

Quality Requirement

Characteristic or attribute of a component of a system. Examples include runtime performance, safety, security, reliability or maintainability. See also software quality.

Quality Tree

(syn: quality attribute utility tree). A hierarchical model to describe product quality: The root "quality" is hierarchically refined in *areas* or topics, which itself are refined again. Quality scenarios form the leaves of this tree.

- Standards for product quality, like ISO 25010, propose *generic* quality trees.
- The quality of a specific system can be described by a *specific* quality tree (see the example below).

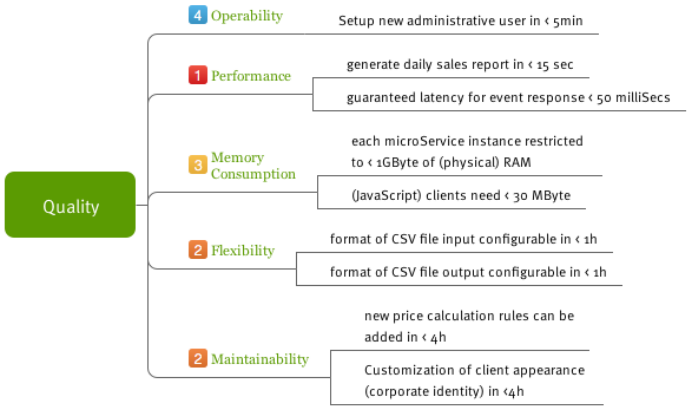

Figure 1. Sample Quality Tree

Quantitative Evaluation

(syn: quantative analysis): Measure or count values of software artifacts, e.g. coupling, cyclomatic complexity, size, test coverage. Metrics like these can help to identify critical parts or elements of systems.

R

Randomness

See Entropy or Pseudo-Randomness.

Rationale

Explanation of the reasoning or arguments that lie behind an architecture decision.

RBAC (Role Based Access Control)

A role is a fixed set of permissions usually assigned to a group of principals. This allows a **Role-Based-Access-Control** usually to be implemented more efficient than an ACL based system and makes for example deputy arrangements possible.

Recoverability Quality Attribute

Degree to which, in the event of an interruption or a failure, a product or system can recover the data directly affected and re-establish the desired state of the system. Is a sub-characteristic of: reliability.

Refer to [ISO-25010].

Redesign

The alteration of software units in such a way that they fulfill a similar purpose as before, but in a different manner and possibly by different means. Often mistakenly called refactoring.

Refactoring

A term denoting the improvement of software units by changing their internal structure without changing the behavior. (see “Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves the internal structure.” – Refactoring, Martin Fowler, 1999) Not to be confused with re**design**.

Registry

"A well-known object that other objects can use to find common objects and services." (quoted from PoEAA). Often implemented as a singleton (also a well-known design pattern).

Relationship

Generic term denoting some kind of dependency between elements of an architecture. Different types of relationship are used within architectures, e.g. call, notification, ownership, containment, creation or inheritance.

Reliability Quality Attribute

Degree to which a system, product or component performs specified functions under specified conditions for a specified period of time. Is composed of the following sub-characteristics: maturity, availability, fault tolerance, recoverability.

Refer to [ISO-25010].

Remote Procedure Call (RPC)

Mechanism for implementing communication between two systems, allowing a program to execute a procedure in another address space, commonly on another physical machine. This communication happens as if it were a local procedure call, abstracting away the complexities of the network communication. RPC is widely used in distributed systems and network applications.

Replaceability Quality Attribute

Degree to which a product can replace another specified software product for the same purpose in the same environment. Is a sub-characteristic of: portability.

Refer to [ISO-25010].

Repository

In architecture documentation: A place where artifacts are stored before an automatic build process collects them into one consistent document.

In domain-driven design: Repository is a building block of domain-driven design. A repository hides technical details of the infrastructure layer to the domain layer. Repositories return entities that are persisted in the database.

Resource Utilization Quality Attribute

Degree to which the amounts and types of resources used by a product or system, when performing its functions, meet requirements. Is a sub-characteristic of: performance efficiency.

Refer to [ISO-25010].

Reusability Quality Attribute

Degree to which an asset can be used in more than one system, or in building other assets. Is a sub-characteristic of: maintainability.

Refer to [ISO-25010].

Reuse/Release Equivalence Principle

A fundamental principle for designing the structure of software systems (also see package principles). It demands that large components are "released" and under version control, in particular if the system uses them from multiple points. Even if we don't release them publicly, we should extract such components from the system and provide them through an external dependency manager with proper version control.

The REP contains two different insights:

1. On the large scale, modularity and low coupling require more than type separation.
2. Reusability of components (even if all "reuse" is internal) translates to overall maintainability.

Risk

Simply said, a risk is the possibility that a problem occurs. A risk involves *uncertainty* about the effects, consequences or implications of an activity or decision, usually with a negative connotation concerning a certain value (such as health, money, or qualities of a system like availability or security).

To quantify a risk the likelihood of occurrence is multiplied by the potential value which is usually a loss – otherwise the risk would be a chance which given the uncertainty might be a potential outcome for some risks.

RM/ODP

The *Reference Model for Open Distributed Processing* is an (abstract) metamodel for documentation of information systems. Defined in ISO/IEC 10746.

Round-trip Engineering

"Concept of being able to make any kind of change to a model as well as to the code generated from that model. The changes always propagate bidirectional and both artifacts are always consistent." (quoted from Wikipedia)

Comment (Gernot Starke)

In my personal opinion, it does not work in practical situations, only in hello-world-like scenarios, as the inverse abstraction (from low-level sourcecode to higher-level architectural elements) usually involves design-decisions and cannot realistically be automated.

Comment (Matthias Bohlen)

Recently I have seen code that originated from DDD where reverse engineering did indeed work.

Ruby

A wonderful programming language.

Runtime View

Shows the cooperation or collaboration of building blocks (respectively their instances) at runtime in concrete scenarios. Should refer to elements of the Building Block View. Could for example (but doesn't need to) be expressed in UML sequence or activity diagrams.

S

S.O.L.I.D. principles

SOLID (single responsibility, open-closed, Liskov substitution, interface segregation and dependency inversion) is an acronym for some principles (named by Robert C. Martin) to improve object-oriented programming and design. The principles make it more likely that a developer will write code that is easy to maintain and extend over time.

For some additional sources, see [SOLID-principles].

Scenario

Quality scenarios document required quality attributes. They "… are brief narratives of expected or anticipated use of a system from both development and end-user viewpoints." ([Kazman+1996]) Thus, they help to describe required or desired qualities of a system in pragmatic and informal manner, yet making the abstract notion of "quality" concrete and tangible.

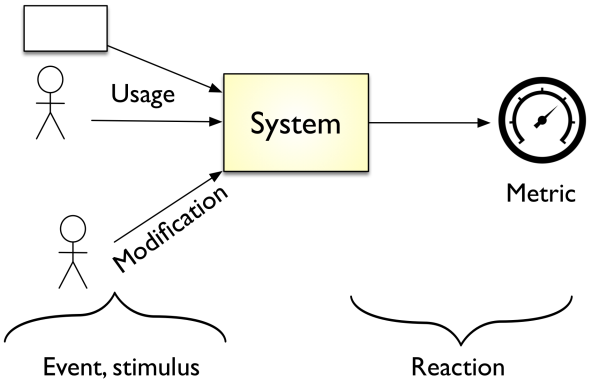

Figure 2. Generic form of (Quality) scenario

- Event/stimulus: Any condition or event arriving at the system
- System (or part of the system) is stimulated by the event.
- Response: The activity undertaken after the arrival of the stimulus.
- Metric (response measure): The response should be measurable in some fashion.

Usually scenarios are differentiated into:

- Usage scenarios (application scenarios)
- Change scenarios (modification or growth scenarios)
- Failure scenarios (boundary, stress, or exploratory scenarios)

SDL

A **Secure-Development-Lifecycle** is a companies usual software development process with additional practices of engineering secure software. This involves for example code reviews, architectural risk analyses, blackbox/whitebox and penetration testing and many more additions. The whole lifecycle of an application should be covered by the SDL, beginning with the first requirements engineering tasks and ending with feedback from operating the released software by fixing security issues.

See [McGraw 2006], page 239.

Security Goals

The goals are the key point of information security. They are a basic set of information attributes which can be fulfilled or not depending on a systems architecture and processes.

The most common agreed set of security goals is the so-called "CIA triad":

- Confidentiality
- Integrity
- Availability

The "Reference Model of Information Assurance and Security" (RMIAS) extends this list by Accountability, Auditability, Authenticity/Trustworthiness, Non-repudiation and Privacy.

These are typical examples for non-functional requirements related to security.

See [Anderson 2008], page 11 or [Cherdantseva+2013].

Security Quality Attribute

Degree to which a product or system protects information and data so that persons or other products or systems have the degree of data access appropriate to their types and levels of authorization. Is composed of the following sub-characteristics: confidentiality, integrity, non-repudiation, accountability, authenticity.

Refer to [ISO-25010].

Self Contained System (SCS)

An architectural style, similar to microservices. To quote from the site scs-architecture.org:

"The Self-contained System (SCS) approach is an architecture that focuses on a separation of the functionality into many independent systems, making the complete system a collaboration of many smaller software systems. This avoids the problem of large monoliths that grow constantly and eventually become unmaintainable"

Sensitivity Point

(in qualitative evaluation like ATAM): Element of the architecture or system influencing several quality attributes. For example, if one component is responsible for both runtime performance *and* robustness, that component is a sensitivity point.

Casually said, if you mess up a sensitivity point, you will most often have more than one problem.

Separation of Concerns (SoC)

Any element of an architecture should have exclusivity and singularity of responsibility and purpose: No element should share the responsibilities of another element or contain unrelated responsibilities.

Another definition is "breaking down a system into elements that overlap as little as possible."

Famous Edgar Dijkstra said in 1974: “Separation of concerns … even if not perfectly possible, is the only available technique for effective ordering of one’s thoughts”.

Similar to the Single Responsibility Principle.

Sequence Diagram

Type of diagram to illustrate how elements of an architecture interact to achieve a certain scenario. It shows the sequence (flow) of messages between elements. As parallel vertical lines it shows the lifespan of objects or components, horizontal lines depict interactions between these components. See the following example.

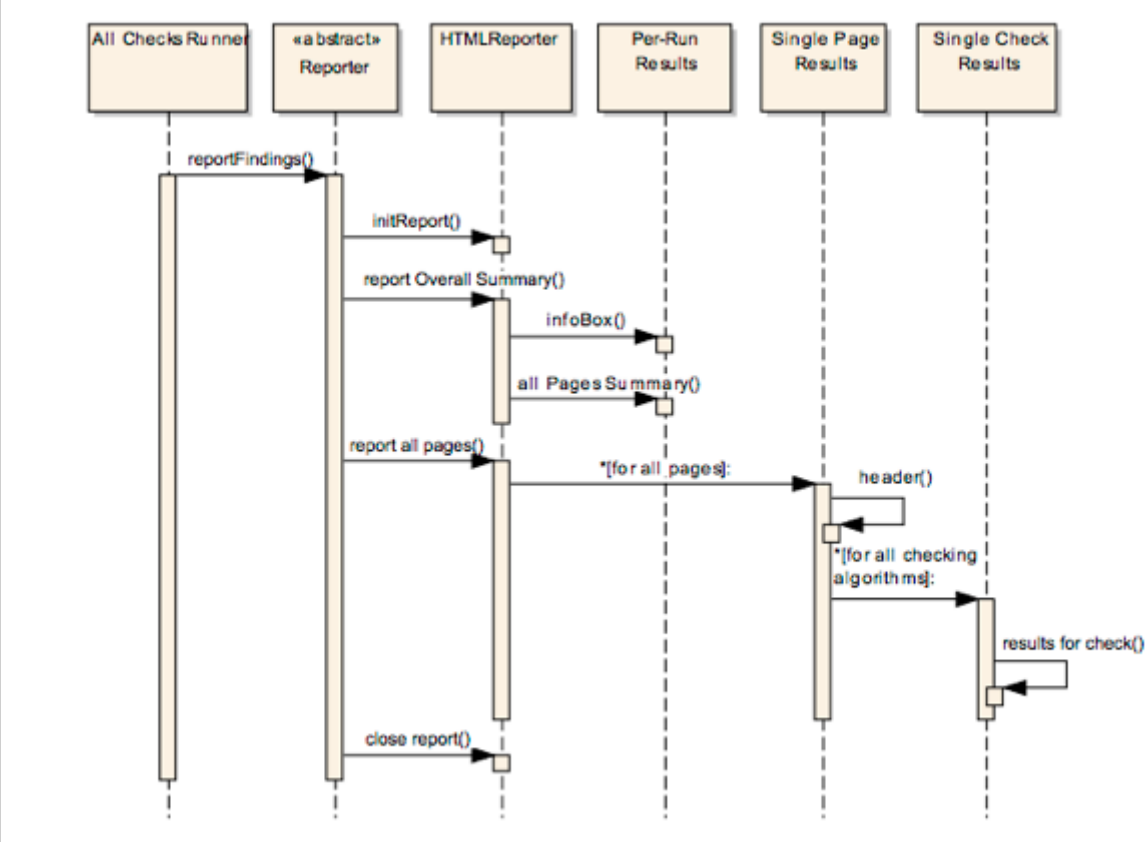

Figure 3. Example of Sequence Diagram

Service

A service is a technical, autonomous unit that bundles related functionalities, typically on a topic, and makes them available for use by other building blocks via a well-defined interface.

A service optimally abstracts the internal, technical function to such an extent that it is not necessary to know or even understand internal implementation details in order to use the service.

Typical examples of externally accessible services are web services, network services, system services or telecommunications services.

(based on Wikipedia)

Service (DDD)

Service is a building block of Domain-driven Design. Services implement logic or processes of the business domain that are not executed by entities alone. A service is stateless and the parameters and return values of its operations are entities, aggregates and value objects.

Service-Oriented Architecture

Architectural style that focuses on providing documented interfaces with interchangeable implementations to users of the system. Services enable reuse across organizational boundaries. See Service.

Signature

Signature of function or method: See function signature.

Digital signature: Method for verifying the authenticity of data or documents.

Single Responsibility Principle (SRP)

Each element within a system or architecture should have a single responsibility, and that all its functions or services should be aligned with that responsibility.

Cohesion is sometimes considered to be associated with the SRP.

Singleton

"Design pattern that restricts the instantiation of a class to one object. This is useful when exactly one object is needed to coordinate actions across the system." (quoted from Wikipedia.)

Software Architecture

There exist several (!) valid and plausible definitions of the term *Software Architecture*.

The following definition has been proposed by the IEEE 1471 standard:

Software Architecture: the fundamental organization of a system embodied in its components, their relationships to each other and to the environment and the principles guiding its design and evolution.

The new standard ISO/IEC/IEEE 42010:2011 has adopted and revised the definition as follows:

Architecture: (system) fundamental concepts or properties of a system in its environment embodied in its elements, relationships, and in the principles of its design and evolution.

The key terms in this definition require some explanation:

- Components: Subsystems, modules, classes, functions or the more general term building blocks: structural elements of software. Components are usually implemented in a programming language, but can also be other artifacts that (together) *make up the system*.
- Relationships: Interfaces, dependencies, associations – different names for the same feature: Components need to interact with other components to enable separation of concerns.

- Environment: Every system has some relationships to its environment: data, control flow or events are transferred to and from maybe different kinds of neighbours.
- Principles: Rules or conventions that hold for a system or several parts of it. Decision or definition, usually valid for several elements of the system. Often called concepts or even *solution patterns*. Principles (concepts) are the foundation for conceptual integrity.

The *Software Engineering Institute* maintains a collection of further definitions

Although the term often refers to the *software architecture of an IT system*, it is also used to refer to *software architecture as an engineering science*.

Software Quality

(from IEEE Standard 1061): Software quality is the degree to which software possesses a desired combination of attributes. This desired combination of attributes need to be clearly defined; otherwise, assessment of quality is left to intuition.

(from ISO/IEC Standard 25010): The quality of a system is the degree to which the system satisfies the stated and implied needs of its various stakeholders, and thus provides value. These stated and implied needs are represented in the ISO 25000 quality models that categorize product quality into characteristics, which in some cases are further subdivided into subcharacteristics.

Stable Abstractions Principle

A fundamental principle for designing the structure of software systems (also see package principles). It demands that the abstractness of components is proportional to their stability. The closely related SDP also explains the notion of **stability** in this context.

We want components that represent abstract concepts and responsibilities to require little or no modification because many conceptually more specific (concrete) components depend on them. And we want components that should or could not easily change to be at least abstract enough so we can extend them. This relates to the OCP.

The SAP can sound like a circular argument until the underlying idea shines through: **Concrete** things and concepts are naturally more volatile, specific, arbitrary and numerous than **abstract** ones. The component structure of a system simply should reflect that. General logic, the system's physical artifacts as well as its functional and technical concepts should all be in congruence.

The SAP is closely related to the SDP. Their combination amounts to a more general and arguably more profound version of the DIP: Specific concepts naturally depend on more **abstract** ones because they are composed or derived from more general-purpose building blocks. And dependent concepts are naturally more **specific** because they are defined by more information than their dependencies (assuming there are no dependence cycles).

Stable Dependencies Principle

A fundamental principle for designing the structure of software systems (also see package principles). It demands that frequently changing components depend on more stable ones.

Part of the volatility of a component is expected and naturally implied by its particular responsibility.

But stability in this context is also a function of incoming and outgoing dependencies. A component that is heavily depended on by others is harder to change and considered to be more stable. A component that

heavily depends on others has more reasons to change and is considered to be less stable.

So with regard to dependence, a component with many clients should not depend on one with many dependencies. A single component with both of these properties is also a red flag. Such a component has many reasons to change but is at the same time hard to change.

Original definitions of the SDP (like [Martin 2003]) involve a metric I of instability. Unfortunately, that metric doesn't capture intended/inherent volatility, transitive dependence or cases like the red flag mentioned above. But we value the idea of the SDP regardless of how it can be measured.

The SDP is closely related to the SAP. Their combination amounts to a version of the DIP (more on this under SAP).

Stakeholder

Person or organization that can be affected by or have an interest (*stake*) in a system, its development or execution.

Examples include users, employees, owners, administrators, developers, designers, project- or product-managers, product-owner, project manager, requirements engineers, business-analysts, government agencies, enterprise architects etc.

Following ISO/IEC/IEEE 42010 a stakeholder is a (system) individual, team, organization, or classes thereof, having an interest in a system (as defined in ISO/IEC/IEEE 42010).

Such interest can be positive (e.g. stakeholder wants to benefit from the system), neutral (stakeholder has to test or verify the system) or negative (stakeholder is competing with the system or wants it to fail).

Strategy

Design pattern for defining a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it. (Quoted from [Gang-of-Four, short: GoF].)

Structural Element

see Building Block or Component

Structure

An arrangement, order or organization of interrelated elements in a system. Structures consist of building blocks (structural elements) and their relationships (dependencies).

Structures in software architecture are often used in architecture views, e.g. the building block view. A documentation template (e.g. arc42) is a kind of structure too.

Sum

Sum data is data that can be one of several distinct, but fixed types. Also known as **discriminated union**, **dijoint union**, **mixed data** or **or data**.

Example:

A **geometric shape** is one of the following:

- a square, or
- a circle, or
- a triangle

Also see [Sperber+2024].

Symmetric Cryptography

Symmetric cryptography is based on an identical key for encryption and decryption of data. Sender and receiver have to agree on a key for communication. See [Schneier 1996], page 17.

System

Collection of elements (building blocks, components etc) organized for a common purpose.

In ISO/IEC/IEEE Standards a couple of system definitions are available:

- systems as described in [ISO/IEC 15288]: “systems that are man-made and may be configured with one or more of the following: hardware, software, data, humans, processes (e.g., processes for providing service to users), procedures (e.g. operator instructions), facilities, materials and naturally occurring entities”.
- software products and services as described in [ISO/IEC 12207].
- software-intensive systems as described in [IEEE Std 1471:2000]: “any system where software contributes essential influences to the design, construction, deployment, and evolution of the system as a whole” to encompass “individual applications, systems in the traditional sense, subsystems, systems of systems, product lines, product families, whole enterprises, and other aggregations of interest”.

System-of-Interest

System-of-Interest (or simply, system) refers to the system whose architecture is under consideration in the preparation of an architecture description (as defined in ISO/IEC/IEEE 42010).

T

Technical Context

Shows the complete system as one blackbox within its environment from a technical or deployment perspective. This includes, in particular, technical interfaces and communication channels as well as the relevant technical details of the neighboring systems. The technical context thus supplements the business context by mapping domain-specific interactions with neighboring systems to e.g. specific communication channels and technical protocols.

See context view.

Template (for Documentation)

Standardized order of artifacts used in software development. It can help base other files, especially documents in a predefines structure without prescribing the content of these single files.

A well known example of such templates is arc42

Template Method

Design pattern that defines the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure. (Quoted from [Gang-of-Four, short: GoF].)

Temporal Coupling

Different interpretations exist from various sources. Temporal coupling

- means that processes that are communicating will both have to be up and running. See [Tanenbaum+2016].
- when you often commit (*modify*) different components at the same time. See [Tornhill 2015].
- when there's an implicit relationship between two, or more, members of a class requiring clients to invoke one member before the other. Mark Seemann, see Design Smell Temporal Coupling.
- means that one system needs to wait for the response of another system before it can continue processing. See Rest Antipattern

Testability Quality Attribute

Degree of effectiveness and efficiency with which test criteria can be established for a system, product or component and tests can be performed to determine whether those criteria have been met. Is a sub-characteristic of: maintainability.

Refer to [ISO-25010].

Time Behaviour Quality Attribute

Degree to which the response and processing times and throughput rates of a product or system, when performing its functions, meet requirements. Is a sub-characteristic of: performance efficiency.

Refer to [ISO-25010].

TLS

Transport-**L**ayer-**S**ecurity is a set of protocols to cryptographically secure the communication of two partys by the means of the CIA-triad. It is widely used for secure communication on the internet and the foundation for HTTPS.

TLS started as an update to its predecessor SSL (Secure Socket Layer) Version 3.0 and should be used now instead of SSL see RFC7568 "Deprecating Secure Sockets Layer Version 3.0".

TOGAF

The Open Group Architecture Framework is a conceptual framework for planning and maintenance of enterprise IT architectures.

Top-Down

"Direction of work" or "order of communication": Starting from an abstract or general construct working towards more concrete, special or detailed representation.

Traceability

(more precisely: *requirements* traceability): Documents that

1. all requirements are addressed by elements of the system (forward traceability) and
2. all elements of the system are justified by at least one requirement (backward traceability).

My personal opinion: If you can, you should avoid traceabiltiy, as it creates a lot of documentation overhead.

Trade-Off

(syn: compromise). A balance achieved or negotiated between two desired or required but usually incompatible or contradicting features. For example, software development usually has to tradeoff memory consumption and runtime speed.

More colloquially, if one thing increases, some other thing must decrease.

Even more colloquially: There is no free lunch. Every quality attribute has a price among other quality attributes.

Trainer

A trainer is a person who conducts a training course himself, provided that this is carried out within the framework of a accreditation granted to an accredited training provider. Accordingly, accredited training providers may only organise and conduct CPSA® training courses with accredited trainers. Only accredited training providers can apply for trainer accreditations.

Training Level

The iSAQB® CPSA® education programme is divided into (currently) two Training Levels: **Foundation Level** and **Advanced Level**. The training levels should contain knowledge that builds upon one another. The exact relationships between each other and the contents of the training level are defined in the respective curricula (*syllabi*).

Training Provider

An organisation or person who holds the rights of use to accredited training materials or who has purchased accreditation for training materials, provides trainers and infrastructure and conducts training courses.

U

Ubiquitous Language

A concept of Domain-driven Design: The ubiquitous language is a language that is structured around the domain model. It is used by all team members to connect all the activities of the team with the software. The ubiquitous language is a living thing that is evolving during a project and will be changed during the whole live cycle of the software.

Unified Modeling Language (UML)

Unified Modeling Language (UML) is a graphical language for visualizing, specifying and documenting the artifacts and structures of a software system.

- For building block views or the context view, use component diagrams, with either components, packages or classes to denote building blocks.
- For runtime views, use sequence- or activity diagrams (with swim-lanes). Object diagrams can

theoretically be used, but are practically not advised, as they become cluttered even for small scenarios.

- For Deployment views, use deployment diagrams with node symbols.

Unit Test

Test of the smallest testable parts of system to determine whether they are fit for use.

Depending on implementation technology, a *unit* might be a method, function, interface or similar element.

Usability Quality Attribute

Degree to which a product or system can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use. Is composed of the following sub-characteristics: appropriateness recognizability, learnability, operability, user error protection, user interface aesthetics, accessibility.

Refer to [ISO-25010].

User Error Protection Quality Attribute

Degree to which a system protects users against making errors. Is a sub-characteristic of: usability.

Refer to [ISO-25010].

User Interface Aesthetics Quality Attribute

Degree to which a user interface enables pleasing and satisfying interaction for the user. Is a sub-characteristic of: usability.

Refer to [ISO-25010].

Uses Relationship

Dependency that exists between two building blocks. If A uses B than execution of A depends on the presence of a correct implementation of B.

V

Value Object

Value Object is a building block of Domain-driven Design. ValueObjects do not have a conceptual identity of their own and should be treated as immutable. They are used to describe the state of entities and may be composed of other value objects but never of entities.

View

See architecture view.

Visitor

Design pattern for representing operations to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates. (Quoted from [Gang-of-Four, short: GoF].)

W

Waterfall Development

Development approach "where you gather all the requirements up front, do all necessary design, down to a detailed level, then hand the specs to the coders, who write the code; then you do testing (possibly with a side trip to IntegrationHell) and deliver the whole thing in one big end-all release. Everything is big including the risk of failure." (quoted from the C2 wiki)

See also iterative development.

Web of Trust

Since a single CA could be an easy target for an attacker the web of trust delegates the establishment of trust to the user. Each user decides which other users proof of identity he trusts, usually by verifying a fingerprint of a given key. This trust is expressed by signing the key of the other user who can then publish it with the additional signature. A third user can then verify this signature and decide to trust the identity or not.

The email encryption PGP is an example for a PKI based on a web of trust.

White Box

Shows the internal structure of a system or building block, made up from blackboxes and the internal/external relationships and interfaces.

See also black box.

Workflow Management System (WFMS)

"Provides an infrastructure for the set-up, performance and monitoring of a defined sequence of tasks, arranged as a workflow." (quoted from Wikipedia)

Wrapper

(syn: Decorator, Adapter, Gateway) Patterns to abstract away the concrete interface or implementation of a component. Attach additional responsibilities to an object dynamically.

Depending on the sources, the semantics of the term *wrapper* may vary.

Comment (Gernot Starke)

The tiny differences found in literature regarding this term often don't matter in real-life. *Wrapping* a component or building-block shall have clear semantics within a single software system.

Translation Tables

Here you find translations of the terms between English and German (see below) and German-to-English (next section).

Several of these terms are based in the legal and organizational foundations of the iSAQB® association (and therefore not related to software architecture).

The following translations are maintained[^generateTranslation] in a simple JSON input file, contained within this books' open source GitHub repository.

[^generateTranslation]: The documentation found in https://github.com/isaqb-org/glossary contains all information required to generate the translation tables. Currently, only English and German are supported. The translation tables are maintained in JSON format, suggestions for improvements are highly welcome!

English to German

Please note: This translation table is not supposed to be complete, several English terms will not be translated but preferably used in their original language (e.g. several pattern names).

English	German
Accessibility	Barrierefreiheit, Zugänglichkeit
Accountability	Rechenschaft, Verantwortlichkeit
Accreditation contract	Akkreditierungsvertrag
Accreditation fee	Akkreditierungsgebühr
Action	Maßnahme
Adaptability	Adaptierbarkeit
Adaption	Anpassung
Adequacy	Angemessenheit
Analysability	Analysierbarkeit
Approach	Ansatz
Appropriateness	Angemessenheit
Appropriateness Recognizability	Erkennbarkeit der Brauchbarkeit, Verständlichkeit
Architectural objective	Architekturziel
Architectural pattern	Architekturmuster
Architectural view	Architektursicht, Sicht
Architecture assessment	Architekturanalyse, Architekturbewertung
Architecture evaluation	Architekturbewertung, Architekturanalyse
Architecture objective	Architekturziel
Articles of assocation	Satzung des Vereins
Artifact	Artefakt
Aspect	Aspekt, Belang

Assessment	Bewertung, Begutachtung, Einschätzung, Untersuchung
Association	Verein, Beziehung
Attack Tree	Angriffsbäume
Authenticity	Authentifizierbarkeit
Availability	Verfügbarkeit
Bounded Context	Kontextgrenze
Building block	Baustein
Building block view	Bausteinsicht
Business	Fachlichkeit, Domäne
Business architecture	fachliche Architektur, Geschäftsarchitektur
Business context	Fachlicher Kontext
Cabinet (as methaphor for template)	Schrank (als Metapher für Template)
Capacity	Kapazität
Cash audit	Rechnungsprüfung
Cash auditor	Rechnungsprüfer
Certification authority	Zertifizierungsstelle
Certification body	Zertifizierungsstelle
Chairman	Vorsitzender
Channel	Kanal
Co-Existence	Koexistenz
Cohesion	Kohäsion, innerer Zusammenhalt
Commensurability	Angemessenheit, Messbarkeit, Vergleichbarkeit
Compatibility	Kompatibilität
Compliance	Erfüllung, Einhaltung
Component	Baustein, Komponente
Concern	Belang
Confidentiality	Vertraulichkeit
Constraint	Randbedingung, Einschränkung
Context (of a term)	Einordnung (eines Begriffes) in einen Zusammenhang
Context view	Kontextabgrenzung
Coupling	Kopplung, Abhängigkeit
Cross-cutting	Querschnittlich
Curriculum	Lehrplan
Decomposition	Zerlegung
Dependency	Abhängigkeit, Beziehung
Deployment	Verteilung
Deployment unit	Verteilungsartefakt

Deployment view	Verteilungssicht
Deputy chairman	Stellvertretender Vorsitzender
Design	Entwurf
Design approach	Entwurfsansatz, Entwurfsmethodik
Design decision	Entwurfsentscheidung
Design principle	Entwurfsprinzip
Domain	Fachdomäne, Fachlicher Bereich, Geschäftsbereich
Domain event	Fachliches Event
Domain-related architecture	fachliche Architektur
Drawing Tool	Mal-/Zeichenprogramm
Economicalness	Sparsamkeit, Wirtschaftlichkeit
Embedded	Eingebettet
Encapsulation	Kapselung
Enterprise IT architecture	Unternehmens-IT-Architektur
Estimation	Schätzung
Evaluation	Bewertung
Examination question	Prüfungsfrage
Examination rules and regulations	Prüfungsordnung
Examination sheet	Prüfungsbogen
Examination task	Prüfungsaufgabe
Examinee	Prüfling
Examiner	Prüfer
Executive board	Vorstand
Fault Tolerance	Fehlertoleranz
Fees rules and regulations	Gebührenordnung
Fitness Function	Fitnessfunktion
Functional Appropriateness	Funktionale Angemessenheit
Functional Completeness	Funktionale Vollständigkeit
Functional Correctness	Funktionale Korrektheit
Functional Suitability	Funktionale Eignung
General meeting	Mitgliederversammlung
Improvement	Verbesserung
Improvement action	Verbesserungsmaßnahme
Influencing Factor	Einflussfaktor
Information hiding principle	Geheimnisprinzip
Installability	Installierbarkeit
Integrity	Integrität
Interdependency (between design decisions)	Abhängigkeit (zwischen Entwurfsentscheidungen)

Interface	Schnittstelle
Interface description	Schnittstellenbeschreibung, Schnittstellendokumentation
Interoperability	Interoperabilität
Learnability	Erlernbarkeit
Learning goal	Lernziel
License fee	Akkreditierungsgebühr
Licensee	Lizenznehmer
Licensing agreement	Lizenzvertrag, Lizenzvereinbarung, Akkreditierungsvertrag
Local court	Amtsgericht
Maintainability	Wartbarkeit
Maturity	Reifegrad
Means for describing	Beschreibungsmittel
Means for documenting	Beschreibungsmittel
Measurability	Messbarkeit
Members' meeting	Mitgliederversammlung
message-driven	Nachrichten-zentrisch
Modeling Tool	Modellierungswerkzeug
Modifiability	Modifizierbarkeit
Modularity	Modularität
Module	Komponente, Modul, Baustein
Node	Knoten
Non-exclusive license	Einfache Lizenz
Non-profit	Gemeinnützig
Non-repudiation	Nichtabstreitbarkeit
Normal case	Normalfall
Notification	Benachrichtigung
Objective	Ziel
Operability	Bedienbarkeit
Operational processes	Betriebsprozesse (von Software)
Pattern	Muster
Pattern language	Mustersprache, Musterfamilie
Performance Efficiency	Leistungseffizienz, Performance
Perspective	Perspektive
Portability	Portierbarkeit
Principle	Prinzip, Konzept
Quality attribute	Qualitätsmerkmal, Qualitätseigenschaft
Quality characteristic	Qualitätsmerkmal, Qualitätseigenschaft

Quality feature	Qualitätsmerkmal, Qualitätseigenschaft
Rationale	Begründung, Erklärung
Real-time system	Echtzeitsystem
Recoverability	Widerherstellbarkeit
Registered trademark	Marke (gesetzlich geschützt)
Relationship	Beziehung
Relationship (kind of)	Beziehungsart
Reliability	Zuverlässigkeit
Replaceability	Austauschbarkeit
Repository	Ablage
Requirement	Anforderung
resilient	unverwüstlich, selbstwiederherstellend
Resolution	Beschluss
Resource Utilization	Ressourcenverbrauch
Responsibility	Verantwortlichkeit
responsive	reaktionsfähig
Reusability	Wiederverwendbarkeit
Rights of use	Nutzungsrecht
Runtime	Laufzeit
Runtime view	Laufzeitsicht
Security	Sicherheit
Security Goals	Schutzziele, Sachziele
Skill	Fähigkeit, Fertigkeit
Specification (of software architecture)	Beschreibung (von Softwarearchitektur)
sponsoring (board) member	materiell förderndes Mitglied
statutory	satzungsgemäß
Structure	Struktur
Task	Aufgabe
Team regulations	Arbeitsgruppenordnung
Technical context	Technischer Kontext
Term	Begriff
Testability	Testbarkeit
Thriftyness	Sparsamkeit, Wirtschaftlichkeit
Time Behaviour	Zeitverhalten
Tools	Arbeitsmittel, Werkzeug
Tools-and-material-approach	Werkzeug-Material-Ansatz
Tradeoff	Kompromiss, Abwägung, Wechselwirkung
Training provider	Schulungsanbieter
Treasurer	Schatzmeister

Ubiquitous language	Allgegenwärtige Sprache
Usability	Benutzbarkeit, Benutzerfreundlichkeit
User Error Protection	Schutz vor Fehlbedienung
User Interface Aesthetics	Ästhetik der Benutzeroberfläche
Uses relationship	Benutzt-Beziehung, Nutzungsbeziehung
View	Sicht, Architektursicht
Workflow management	Ablaufsteuerung
Working environment	Arbeitsumgebung
Working group	Arbeitsgruppe
Working group head	Arbeitsgruppenleiter

German to English

In this section we collect the iSAQB® translation of the terms from German to English.

Please note: This translation table is not supposed to be complete, several English terms will not be translated but preferably used in their original language (e.g. many of the design pattern names).

The actual translation tables are generated from a JSON file in the `/translations` directory.

German	English
Abhängigkeit	Coupling, Dependency
Abhängigkeit (zwischen Entwurfsentscheidungen)	Interdependency (between design decisions)
Ablage	Repository
Ablaufsteuerung	Workflow management
Abwägung	Tradeoff
Adaptierbarkeit	Adaptability
Akkreditierungsgebühr	Accreditation fee, License fee
Akkreditierungsvertrag	Accreditation contract, Licensing agreement
Allgegenwärtige Sprache	Ubiquitous language
Amtsgericht	Local court
Analysierbarkeit	Analysability
Anforderung	Requirement
Angemessenheit	Adequacy, Appropriateness, Commensurability
Angriffsbäume	Attack Tree
Anpassung	Adaption
Ansatz	Approach
Arbeitsgruppe	Working group
Arbeitsgruppenleiter	Working group head
Arbeitsgruppenordnung	Team regulations
Arbeitsmittel	Tools
Arbeitsumgebung	Working environment
Architekturanalyse	Architecture assessment, Architecture evaluation
Architekturbewertung	Architecture assessment, Architecture evaluation
Architekturmuster	Architectural pattern
Architektursicht	Architectural view, View
Architekturziel	Architectural objective, Architecture objective
Artefakt	Artifact
Aspekt	Aspect
Aufgabe	Task
Austauschbarkeit	Replaceability

Authentifizierbarkeit	Authenticity
Barrierefreiheit	Accessibility
Baustein	Building block, Component, Module
Bausteinsicht	Building block view
Bedienbarkeit	Operability
Begriff	Term
Begründung	Rationale
Begutachtung	Assessment
Belang	Aspect, Concern
Benachrichtigung	Notification
Benutzbarkeit	Usability
Benutzerfreundlichkeit	Usability
Benutzt-Beziehung	Uses relationship
Beschluss	Resolution
Beschreibung (von Softwarearchitektur)	Specification (of software architecture)
Beschreibungsmittel	Means for describing, Means for documenting
Betriebsprozesse (von Software)	Operational processes
Bewertung	Assessment, Evaluation
Beziehung	Association, Dependency, Relationship
Beziehungsart	Relationship (kind of)
Domäne	Business
Echtzeitsystem	Real-time system
Einfache Lizenz	Non-exclusive license
Einflussfaktor	Influencing Factor
Eingebettet	Embedded
Einhaltung	Compliance
Einordnung (eines Begriffes) in einen Zusammenhang	Context (of a term)
Einschränkung	Constraint
Einschätzung	Assessment
Entwurf	Design
Entwurfsansatz	Design approach
Entwurfsentscheidung	Design decision
Entwurfsmethodik	Design approach
Entwurfsprinzip	Design principle
Erfüllung	Compliance
Erkennbarkeit der Brauchbarkeit	Appropriateness Recognizability
Erklärung	Rationale
Erlernbarkeit	Learnability

Fachdomäne	Domain
fachliche Architektur	Business architecture, Domain-related architecture
Fachlicher Bereich	Domain
Fachlicher Kontext	Business context
Fachliches Event	Domain event
Fachlichkeit	Business
Fehlertoleranz	Fault Tolerance
Fertigkeit	Skill
Fitnessfunktion	Fitness Function
Funktionale Angemessenheit	Functional Appropriateness
Funktionale Eignung	Functional Suitability
Funktionale Korrektheit	Functional Correctness
Funktionale Vollständigkeit	Functional Completeness
Fähigkeit	Skill
Gebührenordnung	Fees rules and regulations
Geheimnisprinzip	Information hiding principle
Gemeinnützig	Non-profit
Geschäftsarchitektur	Business architecture
Geschäftsbereich	Domain
innerer Zusammenhalt	Cohesion
Installierbarkeit	Installability
Integrität	Integrity
Interoperabilität	Interoperability
Kanal	Channel
Kapazität	Capacity
Kapselung	Encapsulation
Knoten	Node
Koexistenz	Co-Existence
Kohäsion	Cohesion
Kompatibilität	Compatibility
Komponente	Component, Module
Kompromiss	Tradeoff
Kontextabgrenzung	Context view
Kontextgrenze	Bounded Context
Konzept	Principle
Kopplung	Coupling
Laufzeit	Runtime
Laufzeitsicht	Runtime view
Lehrplan	Curriculum

Leistungseffizienz	Performance Efficiency
Lernziel	Learning goal
Lizenznehmer	Licensee
Lizenzvereinbarung	Licensing agreement
Lizenzvertrag	Licensing agreement
Mal-/Zeichenprogramm	Drawing Tool
Marke (gesetzlich geschützt)	Registered trademark
materiell förderndes Mitglied	sponsoring (board) member
Maßnahme	Action
Messbarkeit	Commensurability, Measurability
Mitgliederversammlung	General meeting, Members' meeting
Modellierungswerkzeug	Modeling Tool
Modifizierbarkeit	Modifiability
Modul	Module
Modularität	Modularity
Muster	Pattern
Musterfamilie	Pattern language
Mustersprache	Pattern language
Nachrichten-zentrisch	message-driven
Nichtabstreitbarkeit	Non-repudiation
Normalfall	Normal case
Nutzungsbeziehung	Uses relationship
Nutzungsrecht	Rights of use
Performance	Performance Efficiency
Perspektive	Perspective
Portierbarkeit	Portability
Prinzip	Principle
Prüfer	Examiner
Prüfling	Examinee
Prüfungsaufgabe	Examination task
Prüfungsbogen	Examination sheet
Prüfungsfrage	Examination question
Prüfungsordnung	Examination rules and regulations
Qualitätseigenschaft	Quality attribute, Quality characteristic, Quality feature
Qualitätsmerkmal	Quality attribute, Quality characteristic, Quality feature
Querschnittlich	Cross-cutting
Randbedingung	Constraint

reaktionsfähig	responsive
Rechenschaft	Accountability
Rechnungsprüfer	Cash auditor
Rechnungsprüfung	Cash audit
Reifegrad	Maturity
Ressourcenverbrauch	Resource Utilization
Sachziele	Security Goals
Satzung des Vereins	Articles of assocation
satzungsgemäß	statutory
Schatzmeister	Treasurer
Schnittstelle	Interface
Schnittstellenbeschreibung	Interface description
Schnittstellendokumentation	Interface description
Schrank (als Metapher für Template)	Cabinet (as methaphor for template)
Schulungsanbieter	Training provider
Schutz vor Fehlbedienung	User Error Protection
Schutzziele	Security Goals
Schätzung	Estimation
selbstwiederherstellend	resilient
Sicherheit	Security
Sicht	Architectural view, View
Sparsamkeit	Economicalness, Thriftyness
Stellvertretender Vorsitzender	Deputy chairman
Struktur	Structure
Technischer Kontext	Technical context
Testbarkeit	Testability
Unternehmens-IT-Architektur	Enterprise IT architecture
Untersuchung	Assessment
unverwüstlich	resilient
Verantwortlichkeit	Accountability, Responsibility
Verbesserung	Improvement
Verbesserungsmaßnahme	Improvement action
Verein	Association
Verfügbarkeit	Availability
Vergleichbarkeit	Commensurability
Verständlichkeit	Appropriateness Recognizability
Verteilung	Deployment
Verteilungsartefakt	Deployment unit
Verteilungssicht	Deployment view

Vertraulichkeit	Confidentiality
Vorsitzender	Chairman
Vorstand	Executive board
Wartbarkeit	Maintainability
Wechselwirkung	Tradeoff
Werkzeug	Tools
Werkzeug-Material-Ansatz	Tools-and-material-approach
Widerherstellbarkeit	Recoverability
Wiederverwendbarkeit	Reusability
Wirtschaftlichkeit	Economicalness, Thriftyness
Zeitverhalten	Time Behaviour
Zerlegung	Decomposition
Zertifizierungsstelle	Certification authority, Certification body
Ziel	Objective
Zugänglichkeit	Accessibility
Zuverlässigkeit	Reliability
Ästhetik der Benutzeroberfläche	User Interface Aesthetics

References and Resources

This section contains references that are cited in the glossary or one of the curriculuae.

A

- [Anderson 2008] Ross Anderson, Security Engineering - A Guide to Building Dependable Distributed Systems, 2nd edition 2008, John Wiley & Sons. One of the most comprehensive books about information security available.

B

- [Bachmann+2000] F. Bachmann et al., "Software Architecture Documentation in Practice: Documenting Architectural Layers," Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, USA, Tech. Rep. CMU/SEI-2000-SR-004, Mar. 2000. [Online]. Available: https://insights.sei.cmu.edu/documents/5437/2000_003_001_13649.pdf
- [Bass+2021] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*, 4th ed. Boston, MA, USA: Addison Wesley, 2021. Although the title suggests otherwise, a quite fundamental (and sometimes abstract) book. The authors have a strong background in ultra-large scale (often military) systems - so their advice might sometimes conflict with small or lean kinds of projects.
- [Buschmann+1996] Buschmann, Frank/Meunier, Regine/Rohnert, Hans/Sommerlad, Peter: *A System of Patterns: Pattern-Oriented Software Architecture 1*, 1st edition, 1996, John Wiley & Sons.

Also known as POSA-1. Most likely the most famous and groundbreaking book on architecture patterns.

C

- [Clements+2003] Clements, P., F. Bachmann, L. Bass, D. Garlan, J. Ivers et al.: Documenting Software Architectures – Views and Beyond. Addison Wesley, 2003.
- [Cockburn 2005] Cockburn, Alistair (2005-04-01): Hexagonal architecture, online https://alistair.cockburn.us/hexagonal-architecture/ (retrieved 2024-07-25)

E

- [Evans 2004] Evans, Eric: *Domain-Driven Design: Tackling Complexity in the Heart of Software,* 1st edition, Addison-Wesley, 2004.

F

- [Ford+2017] Neil Ford, Rebecca Parsons, Patrick Kua: Building Evolutionary Architectures: Support Constant Change. OReilly 2017

G

- [GoF: Design-Patterns] Gamma, Erich/Helm, Richard/Johnson, Ralph/Vlissides, John M. *Design Patterns: Elements of Reusable Object-Oriented Software,* 1st edition, 1994, Addison-Wesley, 1994.

A classic on design patterns.

- [Gang-of-Four, short: GoF] See [GoF: Design-Patterns]

H

- [Hargis+2004] Hargis, Gretchen et al.: Quality Technical Information: A Handbook for Writers and Editors. Prentice Hall, IBM Press, 2004.
- [Hofmeister+2000] Hofmeister, Christine/Nord, Robert/Soni, Dilip]]]: *Applied Software Architecture,* 1st edition, Addison-Wesley, 1999
- [Hombergs 2024] Hombergs, Tom: Get Your Hands Dirty on Clean Architecture, Packt, 2nd edition 2024.

I

- [ISO-25010] ISO/IEC 25010:2023(en) Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — Product quality model. Terms and definitions online: https://www.iso.org/obp/ui/#iso:std:iso-iec:25010:ed-2:v1:en
- [ISO-25019] ISO/IEC 25019:2023(en) Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — Quality-in-use model. Terms and definitions online: https://www.iso.org/obp/ui/#iso:std:iso-iec:25019:ed-1:v1:en

K

- [Kazman+1996] Kazman, R., Abowd, G., Bass, L., & Clements, P.: *Scenario-based analysis of software architecture*, IEEE software, 13(6), 47-55, 1996.
- [Kruchten 1995] Kruchten, P.: Architectural Blueprints – The 4-1 View Model of Architecture. IEEE Software November 1995; 12(6), p. 42-50.

L

- [Lange 2021] Kenneth Lange: The Functional Core, Imperative Shell Pattern, online: https://www.kennethlange.com/functional-core-imperative-shell/
- [Lilienthal 2019] Lilienthal, Carola: *Langlebige Software-Architekturen: Technische Schulden analysieren, begrenzen und abbauen* 3rd edition, dpunkt.verlag, 2019

M

- [Maguire 2019] Sandy Maguire: Algebra-Driven Design: Elegant Solutions from Simple Building Blocks. Leanpub, 2019.
- [Martin 2003] Martin, Robert C.: *Agile Software Development: Principles, Patterns and Practices*, Prentice Hall, 2003
- [SOLID-principles] Martin, Robert: SOLID-principles. S.O.L.I.D is an acronym for the first five object-oriented design(OOD) principles by Robert C. Martin. Some original papers have been moved around onto various locations - see Wikipedia
- [McGraw 2006] Garry McGraw, "Software Security - Building Security In", Addison-Wesley 2006 Covering the whole process of software design from a security perspective by the means of risk management, code reviews, risk analysis, penetration testing, security testing abuse case development.

P

- [Parnas 1972] Parnas, David: *On the criteria to be used in decomposing systems into modules"*, Communications of the ACM, volume 15, issue 12, Dec 1972. One of the most influential articles ever written in software engineering, introducing encapsulation and modularity. Thank you, David!

R

- [Cherdantseva+2013] Yulia Cherdantseva, Jeremy Hilton, A Reference Model of Information Assurance & Security, 2013 Eight International Conference on Availability, Reliability and Security (ARES), DOI: 10.1109/ARES.2013.72, http://users.cs.cf.ac.uk/Y.V.Cherdantseva/RMIAS.pdf Conference Paper of Yulia Cherdantseva and Jeremy Hilton describing the RMIAS.
- [Rozanski+2011] Eoin Woods and Nick Rozanski: Software Systems Architecture: Working With Stakeholders Using Viewpoints and Perspectives. 2nd edition 2011, Addison-Wesley. Presents a set of architectural viewpoints and perspectives.

S

- [Schmidt+2000] D. C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann, *Pattern-Oriented Software Architecture, Volume 2: Patterns for Concurrent and Networked Objects*. New York, NY, USA: Wiley & Sons, 2000.
- [Schneier 1996] B. Schneier, *Applied Cryptography*, 2nd ed. New York, NY, USA: John Wiley & Sons, 1996.
- [Sperber+2024] Michael Sperber, Stefan Wehr: Datenmodellierung mit Summen und Produkten, 2024. https://funktionale-programmierung.de/2024/11/25/sums-products.html. (English translation: Data Modeling with Sums and Products, 2024. https://funktionale-programmierung.de/2024/11/25/sums-products-english.html)
- [Starke 2024] G. Starke, *Effektive Software-Architekturen - Ein praktischer Leitfaden*, 10th ed. Munich, Germany: Carl Hanser Verlag, 2024. Website: https://esabuch.de

T

- [Tanenbaum+2016] Andrew Tanenbaum, Maarten van Steen: Distributed Systems, Principles and Paradigms, 2016. https://www.distributed-systems.net/
- [Tornhill 2015] Adam Tornhill: Your Code as a Crime Scene. Use Forensic Techniques to Arrest Defects, Bottlenecks, and Bad Design in Your Programs. Pragmatic Programmers, 2015. https://www.adamtornhill.com

Y

- [Yorgey 2012] Brent A. Yorgey, Monoids: Theme and Variations. Proceedings of the 2012 Haskell Symposium, September 2012 https://doi.org/10.1145/2364506.2364520

Appendix

The iSAQB® e. V. Association

The International Software Architecture Qualification Board (iSAQB® e. V., http://isaqb.org) is a non-profit organization with members from industry, development and consulting firms, education, academia and other organizations.

It is established as an *association* (e. V.) according to German law with the following objectives:

- Creating and maintaining consistent **curricula** for software architects.
- **Defining certification examinations** based upon the various CPSA® curricula
- **Ensuring high quality of teaching** for software architects
- Ensuring a high quality of its software architecture certifications

iSAQB® defines and prescribes training and examination regulations, but does not carry out any training or examinations itself. iSAQB® trainings are carried out by (licensed) training and examination organizations.

iSAQB® monitors and audits the quality of these trainings and all associated processes (e.g. certification procedures).

About the Authors

Gernot Starke

Dr. Gernot Starke (INNOQ Fellow) is co-founder and avid user of the (open source) arc42 documentation template. For more than 20 years he works as software architect, coach and consultant, conquering the challenges of creating effective software architectures for clients from various industries.

In 2008 Gernot co-founded the International Software Architecture Qualification Board (iSAQB® e. V.) and since then supports it as an active member.

Gernot has authored several (German) books on software architecture and related topics and initiated this glossary.

He lives in Cologne with his wife (*Cheffe Uli*).

Ulrich Becker

Ulrich Becker works as principal consultant at Method Park, focussing on software architecture and application lifecycle management.

Ulrich studied computer science at the University of Hamburg and the University of Erlangen-Nürnberg. He received his PhD from the University of Erlangen-Nürnberg in 2003 for his work on model-based distribution configuration. He then became group leader for the adaptive system software group at Fraunhofer IIS.

Since 2005 Ulrich works as a trainer, consultant and coach at Method Park where he supports his clients in improving their development processes and methods. Most of his clients are from the automotive industry or other heavily regulated industries.

Ulrich is a founding member of iSAQB® e. V. where he contributes to the foundation level and advanced level working groups. He lives in Erlangen with his family.

Matthias Bohlen

Matthias Bohlen, independent expert for effective product development, started his career as a software developer in 1980. He wrote compilers for the MC68020 processor by Motorola which was quite a revolutionary device in those days where there was no IBM PC, yet. And the compilers really sold well.

Since then, Matthias has worked with countless software teams, helping them to get working software out the door without losing their mind. This is what he still does today.

Matthias is an active member of the International Software Architecture Qualification Board, writes a blog, is being known in the Lean/Agile field, and speaks at conferences for software development.

Phillip Ghadir

Member of the board of INNOQ Deutschland GmbH. Since many years, Phillip consults clients from various industries in topics around software-architecture, technology and development. He co-founded the iSAQB® and regularly conducts trainings on software architecture.

Carola Lilienthal

Dr. Carola Lilienthal is software architect at and co-founder of the WPS Workplace-Solutions. For 20 years she has been working as a developer, project manager, coach, consultant and architect. Carola was an

early adopter of Domain-Driven Design and agile movement and has successfully worked for numerous clients from various domains, mainly finance, insurance and logistics.

Since 2003, she has been analyzing software systems in Java, C++, C#, PHP, ABAP and gives advice to development teams how to improve the sustainability of their code. Carola speaks regularly on conferences and has written various articles as well as a book on sustainable software architecture.

Since 2008 Carola has been supporting the International Software Architecture Qualification Board (iSAQB® e. V.) as an active member.

Mahbouba Gharbi

Mahbouba Gharbi is CIO of iTech Progress, book author and conference speaker.

Several years ago Mahbouba became president of the iSAQB. She lives in Mannheim with her family.

Simon Kölsch

Simon Kölsch works as a senior consultant at INNOQ with a focus on web architecture and security.

Simon is enthusiastic about solutions beyond the classical monolithic enterprise application, covering the architecture of distributed systems and their infrastructure, logging and monitoring.

He is not committed to one specific technology or programming language, but has a strong JVM background.

Alexander Lorz

Dr.-Ing. Alexander Lorz is a freelance software architecture trainer, IT consultant and developer. His first contact with IT systems dates back to the mid-1980s, and since then he has refused to give up his fascination for the science and craftsmanship of developing complex systems.

As an active member of the International Software Architecture Qualification Board (iSAQB® e. V.) and the Foundation Level Working Group he contributes to the evolution of the foundation curriculum.

Michael Mahlberg

Michael Mahlberg runs his own method consultancy in Germany and spends most of his time supporting clients in their quest for more effective ways to work. Mostly by applying lean and agile concepts.

Running his own computer and software related companies since he was 18, he quickly came to realize that software architecture and (development) processes are in a way timeless aspects of the craft.

Nowadays a lot of his work focuses on processes and human interactions – a field in which he engages both professionally as well as pro bono (for example he is one of the people who started and run the Limited WIP Society Cologne).

Michael's architectural work therefore tends to be dealing with the impact(s) and implications of architectural and process decisions *on each other* and the relative optimization strategies.

Andreas Rausch

Prof. Dr. Andreas Rausch is head of the chair for Software Systems Engineering at Clausthal University of Technology.

He received his doctorate in 2001 at the Technical University of Munich, at the chair of Prof. Dr. Manfred Broy. His main research interests in the field of software systems engineering are software architectures, model-based software development and process models. He has published more than 70 international papers in these areas.

Roger Rhoades

Roger Rhoades is founder of Albion, a training and consulting company in Germany.

Roger has over 25 years of practical experience in the field of enterprise, business, and software architecture as well as management of international teams and projects. This practical experience is integrated into his training courses to ensure that class participants not only understand the theoretical content, but also the real-world challenges of its implementation.

Since 2012, Roger regularly presents at international conferences (e.g. EAMKon, Lean42 EAM, IT Strategy and Governance).

Since 2014, Roger has been an active member of the International Software Architecture Qualification Board (iSAQB® e. V.). He actively supports the evolution of the foundation and advanced curricula, exam questions, and case studies in addition to the iSAQB® glossary.

Sebastian Fichtner

Founder of flowtoolz.com. App engineer and consultant. Started coding in 1995. Passionate about architecture ever since. Does original apps, open source and projects for various clients. Loves Apple platforms and the language Swift.

About our Cause

ELECTRONIC FRONTIER FOUNDATION eff.org

All royalties of this book are donated to the EFF. By paying for this book, you support their cause:

"The **Electronic Frontier Foundation** is the leading nonprofit organization defending civil liberties in the digital world. Founded in 1990, EFF champions user privacy, free expression, and innovation through impact litigation, policy analysis, grassroots activism, and technology development. We work to ensure that rights and freedoms are enhanced and protected as our use of technology grows.

Even in the fledgling days of the Internet, EFF understood that protecting access to developing technology was central to advancing freedom for all. In the years that followed, EFF used our fiercely independent voice to clear the way for open source software, encryption, security research, file sharing tools, and a world of emerging technologies.

Today, EFF uses the unique expertise of leading technologists, activists, and attorneys in our efforts to defend free speech online, fight illegal surveillance, advocate for users and innovators, and support freedom-enhancing technologies.

Together, we forged a vast network of concerned members and partner organizations spanning the globe. EFF advises policymakers and educates the press and the public through comprehensive analysis, educational guides, activist workshops, and more. EFF empowers hundreds of thousands of individuals through our Action Center and has become a leading voice in online rights debates.

EFF is a donor-funded US 501(c)(3) nonprofit organization that depends on your support to continue fighting for users."

(Quote from eff.org/about)