Inofficial Glossary of Software Architecture Terminology for Foundation Level

2022.1-EN, 20230321

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 (Markus Harrer)

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

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.

Be aware that the authors of this version will also mirror your contributions back to the original repository.

Terms

A

Abstraction (activity)

Removing details to focus attention on aspects of greater importance.

Abstraction (result)

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

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.

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.

Appropriateness

(syn: adequacy) Suitability for a particular purpose.

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” ([buschmann1996], page 12). Similar to architecture style.

Examples include:

- Layers
- Pipes-and-Filter
- 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].

Architecture

See Software Architecture

Architecture Description

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

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 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 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] and [rozanski-11] 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).

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.).

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.

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)

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.

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.

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.

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-11], 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.

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.

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.

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.

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*.

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 ([ref-gamma-1994]) 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.

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 [ref-evans-2004].

See also:

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

E

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".

F

Filter

Part of the pipe-and-filter architectural style that creates or transforms data. Filters are typically

executed independently of other filters.

Fitness Function

"An architectural fitness function provides an objective integrity assessment of some architectural characteristics." ([ref-ford-2017]).

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

H

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.

Hybrid Architecture Style

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

I

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.

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.

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.

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 [ref-iso-25010].

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

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.

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*.

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

Microservices

An architectural style, proposing to divide large systems into small units to deliver software systems e.g., faster and scale more cost-effectively.

Model-Driven Software Development (MDSD)

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

Model-View-Controller

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.

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).

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

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*).

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

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*.

P

Pattern

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

See architecture pattern or design pattern.

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-11] 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

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.

Port

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

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).

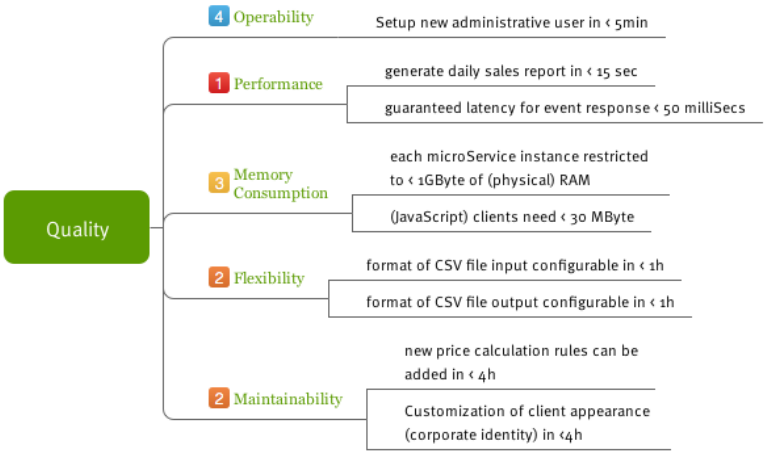

Figure 1. Sample Quality Tree

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

Rationale

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

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**.

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.

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.

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 [martin-solid].

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." ([ref-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.

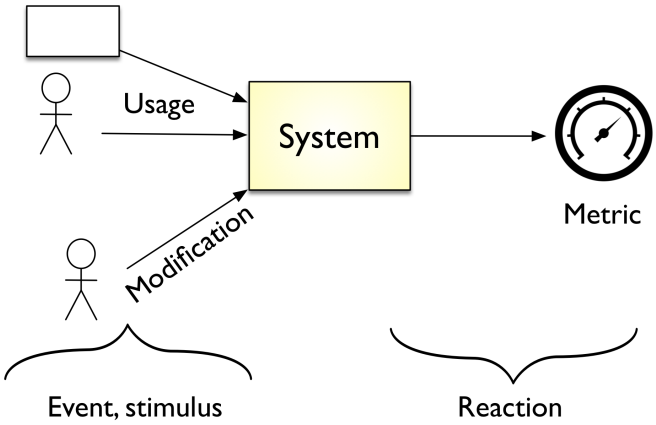

Figure 2. Generic form of (Quality) scenario

- 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)

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.

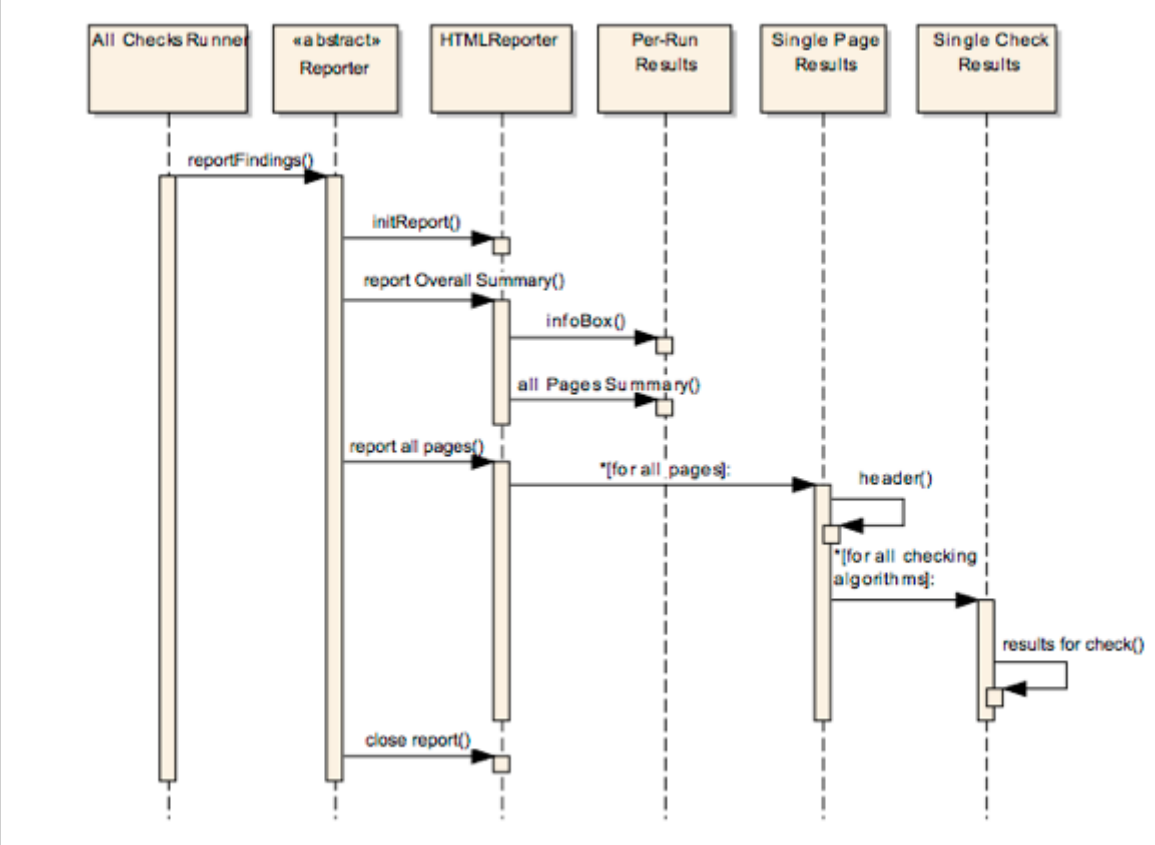

Figure 3. Example of Sequence Diagram

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)

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.

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.

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).

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.

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”.

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

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

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.

U

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.

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

View

See architecture view.

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.

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.

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.