Curriculum for

Certified Professional for Software Architecture (CPSA)®

Foundation Level

2023.1-RC4-EN-20230206

Table of Contents

The abbreviation "e. V." is part of the iSAQB's official name and stands for "eingetragener Verein" (registered association), which describes its status as a legal entity according to German law. For the purpose of simplicity, iSAQB e. V. shall hereafter be referred to as iSAQB without the use of said abbreviation.

List of Learning Goals

- LG 4-4: Quantitative Evaluation of Software Architectures (R2)
- LG 5-1: Know the Relation between Requirements, Constraints, and Solutions (R3)
- LG 5-2: Know the Rationale of a Solution's Technical Implementation (R3)

Introduction

What Does a Foundation Level Training Convey?

Licensed Certified Professional for Software Architecture – Foundation Level (CPSA-F) trainings will provide participants with the knowledge and skills required to design, specify and document a software architecture adequate to fulfil the respective requirements for small and medium-sized systems. Based upon their individual practical experience and existing skills participants will learn to derive architectural decisions from an existing system vision and adequately detailed requirements. CPSA-F trainings teach methods and principles for design, documentation and evaluation of software architectures, independent of specific development processes.

Focus is education and training of the following skills:

- discuss and reconcile fundamental architectural decisions with stakeholders from requirements, management, development, operations and test
- understand the essential activities of software architecture, and carry out those for small- to medium sized systems
- document and communicate software architectures based upon architectural views, architecture patterns and technical concepts.

In addition, such trainings cover:

- the term software architecture and its meaning
- the tasks and responsibilities of software architects
- the roles of software architects within development projects
- state-of-the-art methods and techniques for developing software architectures.

Out of Scope

This curriculum reflects the contents currently considered by the iSAQB members to be necessary and useful for achieving the learning goals of CPSA-F. It is not a comprehensive description of the entire domain of 'software architecture'.

The following topics or concepts are not part of CPSA-F:

- specific implementation technologies, frameworks or libraries
- programming or programming languages
- specific process models
- fundamentals of modelling notations (such as UML) or fundamentals of modelling itself
- system analysis and requirements engineering (please refer to the education and certification program by IREB e. V., https://ireb.org, International Requirements Engineering Board)
- software testing (please refer to the education and certification program by ISTQB e.V., https://istqb.org, International Software Testing Qualification Board)
- project or product management
- introduction to specific software tools.

```
The aim of the training is to provide the basics for acquiring the advanced
knowledge and skills required for the respective application.
```

Prerequisites

The iSAQB e. V. may check the following prerequisites in certification examinations via corresponding questions.

Participants should have the following knowledge and/or experience. In particular, substantial practical experience from software development in a team is an important prerequisite for understanding the learning material and successful certification.

- more than 18 months of practical experience with software development, gained through team-based development of several systems outside of formal education
- knowledge of and practical experience with at least one higher programming language, especially:
  - concepts of
    - modularization (packages, namespaces, etc.)
    - parameter-passing (*call-by-value, call-by-reference*)
    - *scope*, i.e. of type and variable declaration and definition
  - basics of type systems (static vs. dynamic typing, generic data types)
  - error and exception handling in software
  - potential problems of global state and global variables
- Basic knowledge of:
  - modelling and abstraction
  - algorithms and data structures (i.e. Lists, Trees, HashTable, Dictionary, Map)
  - UML (class, package, component and sequence diagrams) and their relation to source code
  - approaches to testing of software (e.g. unit- and acceptance testing)

Furthermore, the following will be useful for understanding several concepts:

- basics and differences of imperative, declarative, object-oriented and functional programming
- practical experience in
  - a higher level programming language
  - designing and implementing distributed applications, such as client-server systems or web applications
  - technical documentation, especially documenting source code, system design or technical concepts

Structure, Duration and Teaching Methods

Study times given in the following sections of the curriculum are just recommendations. The duration of a training course should be at least three days, but may as well be longer. Providers may vary in their approach to duration, teaching methods, the type and structure of exercises as well as the detailed course outline. The types (domains and technologies) of examples and exercises can be determined individually by training providers.

Content	Recommended Duration (min)
1. Basic Concepts of Software Architecture	120
2. Design and Development	420
3. Specification and Communication	240
4. Software Architecture and Quality	120
5. Examples	90
Total	990

Learning Goals and Relevance for the Examination

The structure of the curriculum's chapters follows a set of prioritized learning goals. For each learning goal, relevance for the examination of this learning goal or its sub-elements is clearly stated (by the R1, R2, R3 classification, see the table below). Every learning goal describes the contents to be taught including their key terms and concepts.

Regarding relevance for the examination, the following categories are used in this curriculum:

ID	Learning-goal category	Meaning	Relevance for examination
R1	Being able to	These are the contents participants will be expected to be able to put into practice independently upon completion of the course. Within the course, these contents will be covered through exercises and discussions.	Contents **will be** part of the examination.
R2	Understanding	These are the contents participants are expected to understand in principle. They will normally not be the primary focus of exercises in training.	Contents **may be** part of the examination.
R3	Knowing	These contents (terms, concepts, methods, practices or similar) can enhance understanding and motivate the topic. They may be covered in training if required.	Contents will **not be** part of examination.

If required, the learning goals include references to further reading, standards or other sources. The sections "Terms and Concepts" of each chapter list words that are associated with the contents of the chapter. Some of them are used in the descriptions of learning goals.

Current Version and Public Repository

You find the most current version of this document on the official download page on https://isaqb-org.github.io/.

The document is maintained in a public repository at https://github.com/isaqb-org/curriculum-foundation, all changes and modifications are public.

Please report any issues in our public issue tracker on https://github.com/isaqb-org/curriculum-foundation/issues.

1. Basic Concepts of Software Architecture

Duration: 120 min.	Exercises: none

Relevant Terms

Software architecture; architecture domains; structure; building blocks; components; interfaces; relationships; cross-cutting-concepts; software architects and their responsibilities; tasks and required skills; stakeholders and their concerns; requirements; constraints; influencing factors; types of IT systems (embedded systems; real-time systems; information systems etc.)

Learning Goals

LG 1-1: Discuss Definitions of Software Architecture (R1)

Software architects know several definitions of software architecture (incl. ISO 42010/IEEE 1471, SEI, Booch etc.) and can name their similarities:

- components/building blocks with interfaces and relationships
- building blocks as a general term, components as a special form thereof
- structures, cross-cutting concepts, principles
- architecture decisions and their consequences on the entire systems and its lifecycle

LG 1-2: Understand and Explain the Goals and Benefits of Software Architecture (R1)

Software architects can justify the following essential goals and benefits of software architecture:

- support the design, implementation, maintenance, and operation of systems
- achieve functional requirements or ensure that they can be met
- achieve requirements such as reliability, maintainability, changeability, security, energy efficiency etc.
- ensure that the the system’s structures and concepts are understood by all relevant stakeholders
- systematically reduce complexity
- specify architecturally relevant guidelines for implementation and operation

LG 1-3: Understand Software Architecture as Part of the Software Lifecycle (R2)

Software architects understand their tasks and can integrate their results into the overall lifecycle of IT systems. They can:

- identify the consequences of changes in requirements, technologies, or the system environment in relation to software architecture
- elaborate on relationships between IT-systems and the supported business and operational processes

LG 1-4: Understand Software Architects' Tasks and Responsibilities (R1)

Software architects are responsible for meeting requirements and creating the architecture design of a solution. Depending on the actual approach or process model used, they must align this responsibility with

the overall responsibilities of project management and/or other roles.

Tasks and responsibilities of software architects:

- clarify and scrutinize requirements and constraints, and refine them if necessary, including *required features* and *required constraints*
- decide how to decompose the system into building blocks, while determining dependencies and interfaces between the building blocks
- determine and decide on cross-cutting concepts (for instance persistence, communication, GUI etc.)
- communicate and document software architecture based on views, architectural patterns, cross-cutting and technical concepts
- accompany the realization and implementation of the architecture; integrate feedback from relevant stakeholders into the architecture if necessary; review and ensure the consistency of source code and software architecture
- analyze and evaluate software architecture, especially with respect to risks that involve meeting the requirements, see LG 4-3: Qualitative Analysis of Software Architectures (R2-R3) and LG 4-4: Quantitative Evaluation of Software Architectures (R2),
- identify, highlight, and justify the consequences of architectural decisions to other stakeholders

They should independently recognize the necessity of iterations in all tasks and point out possibilities for appropriate and relevant feedback.

LG 1-5: Relate the Role of Software Architects to Other Stakeholders (R1)

Software architects are able to explain their role. They should adapt their contribution to a software development in a specific context and in relation to other stakeholders and organizational units, in particular to:

- product management and product owners
- project managers
- requirement engineers (requirements- or business analysts, requirements managers, system analysts, business owners, subject-matter experts, etc.)
- developers
- quality assurance and testers
- IT operators and administrators (applies primarily to production environment or data centers for information systems),
- hardware developers
- enterprise architects and architecture board members

LG 1-6: Can Explain the Correlation between Development Approaches and Software Architecture (R2)

- software architects are able to explain the influence of iterative approaches on architectural decisions (with regard to risks and predictability).
- due to inherent uncertainty, software architects often have to work and make decisions iteratively. To do so, they have to systematically obtain feedback from other stakeholders.

LG 1-7: Differentiate between Short- and Long-Term Goals (R2)

Software architects can explain potential conflicts between short-term and long-term goals, in order to find a suitable solution for all stakeholders

LG 1-8: Distinguish Explicit Statements and Implicit Assumptions (R1)

Software architects:

- should explicitly present assumptions or prerequisites, therefore avoiding implicit assumptions
- know that implicit assumptions can lead to potential misunderstandings between stakeholders
- can formulate implicitly, if it is appropriate in the given context

LG 1-9: Responsibilities of Software Architects within the Greater Architectural Context (R3)

The focus of the iSAQB CPSA Foundation Level is on structures and concepts of individual software systems.

In addition, software architects are familiar with other architectural domains, for example:

- enterprise IT architecture: Structure of application landscapes
- business and process architecture: Structure of, among other things, business processes
- information architecture: cross-system structure and use of information and data
- infrastructure or technology architecture: Structure of the technical infrastructure, hardware, networks, etc.
- hardware or processor architecture (for hardware-related systems)

These architectural domains are not the content focus of CPSA-F.

LG 1-10: Differentiate Types of IT Systems (R3)

Software architects know different types of IT systems, for example:

- information systems
- decision support, data warehouse or business intelligence systems
- mobile systems
- *cloud native* systems (refer to [Cloud-Native])
- batch processes or systems
- systems based upon machine learning or artificial intelligence
- hardware-related systems; here they understand the necessity of hardware/software co-design (temporal and content-related dependencies of hardware and software design)

LG 1-11: Challenges of Distributed Systems (R3)

Software architects are able to:

- identify distribution in a given software architecture

- analyze consistency criteria for a given business problem
- explain causality of events in a distributed system

Software architects know:

- communication may fail in a distributed system
- limitations regarding consistency in real-world databases
- what the "split-brain" problem is and why it is difficult
- that it is impossible to determine the temporal order of events in a distributed system

References

[Bass+ 2021], [Gharbi+2020], [iSAQB References], [Lorz+2021], [Starke 2020], [vanSteen+Tanenbaum]

2. Design and Development of Software Architectures

Duration: 330 min.	Excercises: 90 min.

Relevant Terms

Design; design approach; design decision; views; interfaces; technical and cross-cutting concepts; architectural patterns; pattern languages; design principles; dependencies; coupling; cohesion; top-down and bottom-up approaches; model-based design; iterative/incremental design; domain-driven design

Learning Goals

LG 2-1: Select and Use Approaches and Heuristics for Architecture Development (R1,R3)

Software architects are able to name, explain, and use fundamental approaches of architecture development, for example:

- top-down and bottom-up approaches to design (R1)
- view-based architecture development (R1)
- iterative and incremental design (R1)
  - necessity of iterations, especially when decision-making is affected by uncertainties (R1)
  - necessity of feedback on design decisions (R1)
- domain-driven design, see [Evans 2004] (R3)
- evolutionary architecture, see [Ford 2017] (R3)
- global analysis, see [Hofmeister et. al 1999] (R3)
- model-driven architecture (R3)

LG 2-2: Design Software Architectures (R1)

Software architects are able to:

- design and appropriately communicate and document software architectures based upon known functional and quality requirements for software systems that are neither safety- nor business-critical
- make structure-relevant decisions regarding system decomposition and building-block structure and deliberately design dependencies between building blocks
- recognize and justify interdependencies and trade-offs of design decisions
- explain the terms *black box* and *white box* and apply them purposefully
- apply stepwise refinement and specify building blocks
- design architecture views, especially building-block view, runtime view and deployment view
- explain the consequences of these decisions on the corresponding source code
- separate technical and domain-related elements of architectures and justify these decisions
- identify risks related to architecture decisions.

LG 2-3: Identify and Consider Factors Influencing Software Architecture (R1-R3)

Software architects are able to clarify and consider requirements (including constraints that restrict their decisions). They understand that their decisions can lead to additional requirements (including constraints) on the system being designed, its architecture, or the development process.

They should recognize and account for the impact of:

- product-related factors and trends such as (R1)
  - functional requirements
  - quality requirements
  - technological, organizational and regulatory constraints.
- technological constraints such as
  - existing or planned hardware and software infrastructure (R1)
  - technological constraints on data structures and interfaces (R2)
  - reference architectures, libraries, components, and frameworks (R1)
  - programming languages (R2)
- organizational constraints such as
  - organizational structure of the development team and of the customer (R1), in particular Conway's law (R2).
  - company and team cultures (R3)
  - partnerships and cooperation agreements (R2)
  - standards, guidelines, and process models (e.g. approval and release processes) (R2)
  - available resources like budget, time, and staff (R1)
  - availability, skill set, and commitment of staff (R1)
- regulatory constraints such as (R2)
  - local and international legal constraints
  - contract and liability issues
  - data protection and privacy laws
  - compliance issues or obligations to provide burden of proof
- trends such as (R3)
  - market trends
  - technology trends (e.g. blockchain, microservices)
  - methodology trends (e.g. agile)
  - (potential) impact of further stakeholder concerns and mandated design decisions

Software architects are able to describe how those factors can influence design decisions and can elaborate on the consequences of changing influencing factors by providing examples for some of them (R2).

LG 2-4: Design and Implement Cross-Cutting Concepts (R1)

Software architects are able to:

- explain the significance of such cross-cutting concepts
- decide on and design cross-cutting concepts, for example persistence, communication, GUI, error handling, concurrency, energy efficiency
- identify and assess potential interdependencies between these decisions.

Software architects know that such cross-cutting concepts may be re-used throughout the system.

LG 2-5: Describe, Explain and Appropriately Apply Important Solution Patterns (R1, R3)

Software architects know:

- various architectural patterns and can apply them when appropriate
- that patterns are a way to achieve certain qualities for given problems and requirements within given contexts
- that various categories of patterns exist (R3)
- additional sources for patterns related to their specific technical or application domain (R3)

Software architects can explain and provide examples for the following patterns (R1):

- *layers*:
  - abstraction layers hide details, example: ISO/OSI network layers, or "hardware abstraction layer". See https://en.wikipedia.org/wiki/Hardware_abstraction
  - another interpretation are Layers to (physically) separate functionality or responsibility, see https://en.wikipedia.org/wiki/Multitier_architecture
- *pipes and filter*: representative for data flow patterns, breaking down stepwise processing into a series of processing-activities ("Filter") and associated data transport/buffering capabilities ("Pipes").
- *microservices* split application into separate executable that communicate via network
- *dependency injection* as a possible solution for the Dependency-Inversion-Principle [Newman 2015]

Software architects can explain several of the following patterns, explain their relevance for concrete systems, and provide examples. (R3)

- *blackboard*: handle problems that cannot be solved by deterministic algorithms but require diverse knowledge
- *broker*: responsible for coordinating communication between provider(s) and consumer(s), applied in distributed systems. Responsible for forwarding requests and/or transmitting results and exceptions
- *combinator* (synonym: closure of operations), for domain object of type T, look for operations with both input and output type T. See [Yorgey 2012]
- *CQRS* (Command-Query-Responsibility-Segregation): separates read from write concerns in information systems. Requires some context on database-/persistence technology to understand the different properties and requirements of "read" versus "write" operations
- *event sourcing*: handle operations on data by a sequence of events, each of which is recorded in an

append-only store

- *interpreter*: represent domain object or DSL as syntax, provide function implementing a semantic interpretation of domain object separately from domain object itself
- integration and messaging patterns (e.g. from Hohpe+2004])
- the MVC (Model View Controller), MVVM (Model View ViewModel), MVU (Model View Update), PAC (Presentation Abstraction Control) family of patterns, separating external representation (view) from data, services and their coordination
- interfacing patterns like Adapter, Facade, Proxy. Such patterns help in integration of subsystems and/or simplification of dependencies. Architects should know that these patterns can be used independent of (object) technology
  - *adapter*: decouple consumer and provider - where the interface of the provider does not exactly match that of the consumer. The Adapter decouples one party from interface-changes in the other
  - *facade*: simplifies usage of a provider for consumer(s) by providing simplified access
  - *proxy*: an intermediate between consumer and provider, enabling temporal decoupling, caching of results, controlling access to the provider etc.
- *observer*: a producer of values over time notifies a central switchboards where consumers can register to be notified of them
- *plug-in*: extend the behaviour of a component
- *ports & adapters* (syn. Onion-Architecture, Hexagonal-Architecture, Clean-Architecture): concentrate domain logic in the center of the system, have connections to the outside world (database, UI) at the edges, dependencies only outside-in, never inside-out [Lange 2021] [Martin 2017]
- *remote procedure call*: make a function or algorithm execute in a different address space
- *SOA* (Service-Oriented Architecture): an approach to provide abstract services rather than concrete implementations to users of the system to promote reuse of services across departments and between companies
- *template and strategy*: make specific algorithms flexible by encapsulating them
- *visitor*: separate data-structure traversal from specific processing

Software architects know essential sources for architectural patterns, such as POSA (e.g. [Buschmann+ 1996]) and PoEAA ([Fowler 2002]) (for information systems) (R3).

LG 2-6: Explain and Use Design Principles (R1-R3)

Software architects are able to explain what design principles are. They can outline their general objectives and applications with regard to software architecture. (R2)

Software architects are able to:

- explain the design principles listed below and can illustrate them with examples
- explain how those principles are to be applied
- explain how the requirements determine which principles should be applied
- explain the impact of design principles on the implementation
- analyze source code and architecture designs to evaluate whether these design principles have been

applied or should be applied

Abstraction (R1)

- in the sense of a means for deriving useful generalizations
- as a design technique, where building blocks are dependent on the abstractions rather than depending on implementations
- interfaces as abstractions

Modularization (R1-R3)

- information hiding and encapsulation (R1)
- separation of concerns - SoC (R1)
- loose, but functionally sufficient, coupling (R1) of building blocks, see LG 2-7
- high cohesion (R1)
- SOLID principles (R1-R3), which have, to a certain extent, relevance at the architectural level
  - S: Single responsibility principle (R1) and its relation to SoC
  - O: Open/closed principle (R1)
  - L: Liskov substitution principle (R3) as a way to achieve consistency and conceptual integrity in OO design
  - I: Interface segregation principle (R2), including its relation to LG 2-9
  - D: Dependency inversion principle (R1) by means of interfaces or similar abstractions

Conceptual integrity (R2)

- meaning uniformity (homogeneity, consistency) of solutions for similar problems (R2)
- as a means to achieve the principle of least surprise (R3)

Simplicity (R1-R2)

- as a means to reduce complexity (R1)
- as the driving factor behind KISS (R3) and YAGNI (R3)

Expect errors (R1-R2)

- as a means to design for robust and resilient systems (R1)
- as a generalization of the robustness principle (*Postel's law*) (R2)

Other principles (R3)

Software architects know other principles (such as CUPID, see [Nygard 2022]), and can apply them.

LG 2-7: Manage Dependencies between Building Blocks (R1)

Software architects understand dependencies and coupling between building blocks and can use them in a targeted manner. They:

- know and understand different types of dependencies of building blocks (e.g. coupling via use/delegation, messaging/events, composition, creation, inheritance, temporal coupling, coupling via data, data types or hardware)
- understand how dependencies increase coupling
- can use such types of coupling in a targeted manner and can assess the consequences of such dependencies
- know and can apply possibilities to reduce or eliminate coupling, for example:
  - patterns (see LG 2-5)
  - fundamental design principles (see LG 2-6)
  - externalization of dependencies, i.e. defining concrete dependencies at installation- or runtime, for example by using *Dependency Injection*.

LG 2-8: Achieve Quality Requirements with Appropriate Approaches and Techniques (R1)

Software architects understand and consider the considerable influence of quality requirements in architecture and design decisions, e.g. for:

- efficiency, runtime performance
- availability
- maintainability, modifiability, extensibility, adaptability
- energy efficiency

They can:

- explain and apply solution options, *Architectural Tactics*, suitable practices as well as technical possibilities to achieve important quality requirements of software systems (different for embedded systems or information systems)
- identify and communicate possible trade-offs between such solutions and their associated risks

LG 2-9: Design and Define Interfaces (R1-R3)

Software architects know about the importance of interfaces. They are able to design or specify interfaces between architectural building blocks as well as external interfaces between the system and elements outside of the system.

They know:

- desired characteristics of interfaces and can use them in the design:
  - easy to learn, easy to use, easy to extend
  - hard to misuse
  - functionally complete from the perspective of users or building blocks using them.
- the necessity to treat internal and external interfaces differently
- different approaches for implementing interfaces (R3):
  - resource oriented approach (REST, Representational State Transfer)

- service oriented approach (see WS-*/SOAP-based web services.

LG 2-10: Know Fundamental Principles of Software Deployment (R3)

Software architects:

- know that software deployment is the process of making new or updated software available to its users
- are able to name and explain fundamental concepts of software deployment, for example:
  - automated deployments
  - repeatable builds
  - consistent environments (e.g. use immutable and disposable infrastructure)
  - put everything under version-control
  - releases are easy-to-undo

References

[Bass+ 2021], [Fowler 2002], [Gharbi+2020], [Gamma+94], [Martin 2003], [Buschmann+ 1996], [Buschmann+ 2007], [Starke 2020], [Lilienthal 2019], [Lorz+2021]

3. Specification and Communication of Software Architectures

Duration: 180 min.	Exercises: 60 min.

Relevant Terms

(Architectural) Views; structures; (technical) concepts; documentation; communication; description; stakeholder-oriented, meta structures and templates for description and communication; system context; building blocks; building-block view; runtime view; deployment view; node; channel; deployment artifacts; mapping building blocks onto deployment artifacts; description of interfaces and design decisions; UML, tools for documentation

Learning Goals

LG 3-1: Explain and Consider the Requirements of Technical Documentation (R1)

Software architects know the requirements of technical documentation and can consider and fulfil them when documenting systems:

- understandability, correctness, efficiency, appropriateness, maintainability
- form, content, and level of detail tailored to the stakeholders

They know that only the target audiences can assess the understandability of technical documentation.

LG 3-2: Describe and Communicate Software Architectures (R1, R3)

Software architects use documentation to support the design, implementation and further development (also called *maintenance* or *evolution*) of systems (R2)

Software architects are able to (R1):

- document and communicate architectures for corresponding stakeholders, thereby addressing different target groups, e.g. management, development teams, QA, other software architects, and possibly additional stakeholders
- consolidate and harmonise the style and content of contributions from different groups of authors
- develop and implement measures to support the convergence of written and verbal communication, and balance one against the other appropriately

Software architects know (R1):

- the benefits of template-based documentation
- that various properties of documentation depend on specific properties of the system, its requirements, risks, development process, organization or other factors.

For example, software architects can adjust the following documentation characteristics according to the situation (R3):

- the amount and level of detail of documentation needed
- the documentation format

- the accessibility of the documentation
- formality of documentation (e.g. diagrams compliant to a meta model or simple drawings)
- formal reviews and sign-off processes for documentation

LG 3-3: Explain and Apply Notations/Models to Describe Software Architecture (R2-R3)

Software architects know at least the following UML (see [UML]) diagrams to describe architectural views:

- class, package, component (all R2) and composite-structure diagrams (R3)
- deployment diagrams (R2)
- sequence and activity diagrams (R2)
- state machine diagrams (R3)

Software architects know alternative notations to UML diagrams, for example: (R3)

- Archimate, see [Archimate]
- for runtime views for example flow charts, numbered lists or business-process-modeling-notation (BPMN).

LG 3-4: Explain and Use Architectural Views (R1)

Software architects are able to use the following architectural views:

- context view
- building block or component view (composition of software building blocks)
- runtime view (dynamic view, interaction between software building blocks at runtime, state machines)
- deployment view (hardware and technical infrastructure as well as the mapping of software building blocks onto the infrastructure)

LG 3-5: Explain and Apply Context View of Systems (R1)

Software architects are able to:

- depict the context of systems, e.g. in the form of context diagrams with explanations
- represent external interfaces of systems in the context view
- differentiate business and technical context.

LG 3-6: Document and Communicate Cross-Cutting Concepts (R2)

Software architects are able to adequately document and communicate typical cross-cutting concepts (synonym: *principles*, *aspects*), e. g., persistence, workflow management, UI, deployment/integration, logging.

LG 3-7: Describe Interfaces (R1)

Software architects are able to describe and specify both internal and external interfaces.

LG 3-8: Explain and Document Architectural Decisions (R1-R2)

Software architects are able to:

- systematically take, justify, communicate, and document architectural decisions
- identify, communicate, and document interdependencies between design decisions

Software architects know about Architecture-Decision-Records (ADR, see [Nygard 2011]) and can apply these to document decisions (R2).

LG 3-9: Know Additional Resources and Tools for Documentation (R3)

Software architects know:

- basics of several published frameworks for the description of software architectures, for example:
  - ISO/IEEE-42010 (formerly 1471), see [ISO 42010]
  - arc42, see [arc42]
  - C4, see [Brown]
  - FMC, see [FMC]
- ideas and examples of checklists for the creation, documentation, and testing of software architectures
- possible tools for creating and maintaining architectural documentation

References

[arc42], [Archimate], [Bass+ 2021], [Brown], [Clements+ 2010], [FMC], [Gharbi+2020], [Lorz+2021], [Nygard 2011], [Starke 2020], [UML], [Zörner 2021]

4. Software Architecture and Quality

Duration: 60 min.	Exercises: 60 min.

Relevant Terms

Quality; quality characteristics (also called quality attributes); DIN/ISO 25010; quality scenarios; quality tree; trade-offs between quality characteristics; qualitative architecture analysis; metrics and quantitative analysis

Learning Goals

LG 4-1: Discuss Quality Models and Quality Characteristics (R1)

Software architects can explain:

- the concept of quality and quality characteristics (based on [ISO 25010])
- generic quality models (such as [ISO 25010])
- correlations and trade-offs of quality characteristics, for example:
  - configurability versus reliability
  - memory requirements versus performance efficiency
  - security versus usability
  - runtime flexibility versus maintainability.

LG 4-2: Clarify Quality Requirements for Software Architectures (R1)

Software architects can:

- clarify and formulate specific quality requirements for the software to be developed and its architectures, for example in the form of scenarios and quality trees
- explain and apply scenarios and quality trees.

LG 4-3: Qualitative Analysis of Software Architectures (R2-R3)

Software architects:

- know methodical approaches for the qualitative analysis of software architectures (R2), for example, as specified by ATAM (R3);
- can qualitatively analyze smaller systems (R2)
- know that the following sources of information can help in the qualitative analysis of architectures (R2):
  - quality requirements, e.g. in the form of quality trees and scenarios
  - architecture documentation
  - architecture and design models
  - source code

- metrics
- other documentation of the system, such as requirements, operational or test documentation.

LG 4-4: Quantitative Evaluation of Software Architectures (R2)

Software architects know approaches for the quantitative analysis and evaluation (measurement) of software.

They know that:

- quantitative evaluation can help to identify critical parts within systems
- further information can be helpful for the evaluation of architectures, for example:
  - requirements and architecture documentation
  - source code and related metrics such as lines of code, (cyclomatic) complexity, inbound and outbound dependencies
  - known errors in source code, especially error clusters
  - test cases and test results.
- the use of a metric as a target can lead to its invalidation (R2), as described, e.g., by Goodhart's law (R3).

References

[Bass+ 2021], [Clements+ 2002], [Gharbi+2020], [Lorz+2021], [Martin 2003], [Starke 2020]

5. Examples of Software Architectures

Duration: 90 min.	Exercises: none

This section is not relevant for for the exam.

Learning Goals

LG 5-1: Know the Relation between Requirements, Constraints, and Solutions (R3)

Software architects are expected to recognize and comprehend the correlation between requirements and constraints, and the chosen solutions using at least one example.

LG 5-2: Know the Rationale of a Solution's Technical Implementation (R3)

Software architects understand the technical realization (implementation, technical concepts, products used, architectural decisions, solution strategies) of at least one solution.

References

- [arc42] arc42, the open-source template for software architecture communication, online: https://arc42.org. Maintained on https://github.com/arc42
- [Archimate] The ArchiMate® Enterprise Architecture Modeling Language, online: https://www.opengroup.org/archimate-forum/archimate-overview
- [Bass+ 2021] Len Bass, Paul Clements, Rick Kazman: Software Architecture in Practice. 4th Edition, Addison Wesley 2021.
- [Brown] Simon Brown: Brown, Simon: The C4 model for visualising software architecture. https://c4model.com https://www.infoq.com/articles/C4-architecture-model.
- [Buschmann+ 1996] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stal: Pattern-Oriented Software Architecture (POSA): A System of Patterns. Wiley, 1996.
- [Buschmann+ 2007] Frank Buschmann, Kevlin Henney, Douglas C. Schmidt: Pattern-Oriented Software Architecture (POSA): A Pattern Language for Distributed Computing, Wiley, 2007.
- [Clements+ 2002] Paul Clements, Rick Kazman, Mark Klein: Evaluating Software Architectures. Methods and Case Studies. Addison Wesley, 2002.
- [Clements+ 2010] Paul Clements, Felix Bachmann, Len Bass, David Garlan, David, James Ivers, Reed Little, Paulo Merson and Robert Nord. *Documenting Software Architectures: Views and Beyond*, 2nd edition, Addison Wesley, 2010
- [Cloud-Native] The Cloud Native Computing Foundation, online: https://www.cncf.io/
- [Evans 2004] Eric Evans: *Domain-Driven Design: Tackling Complexity in the Heart of Software,* Addison-Wesley, 2004.
- [FMC] Siegfried Wendt: Fundamental Modeling Concepts, online: http://www.fmc-modeling.org/
- [Ford 2017] Neil Ford, Rebecca Parsons, Patrick Kua: Building Evolutionary Architectures: Support Constant Change. OReilly 2017
- [Fowler 2002] Martin Fowler: Patterns of Enterprise Application Architecture. (PoEAA) Addison-Wesley, 2002.
- [Gharbi+2020] Mahbouba Gharbi, Arne Koschel, Andreas Rausch, Gernot Starke: Basiswissen Softwarearchitektur. 4. Auflage, dpunkt Verlag, Heidelberg 2020.
- [Geirhos 2015] Matthias Geirhos. Entwurfsmuster: Das umfassende Handbuch (in German). Rheinwerk Computing Verlag. 2015
- [Gamma+94] Erich Gamma, Richard Helm, Ralph Johnson & John Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley. 1994.
- [Goll 2014] Joachim Goll: Architektur- und Entwurfsmuster der Softwaretechnik: Mit lauffähigen Beispielen in Java (in German). Springer-Vieweg Verlag, 2. Auflage 2014.
- [Hofmeister et. al 1999] Christine Hofmeister, Robert Nord, Dilip Soni: *Applied Software Architecture*, Addison-Wesley, 1999
- [ISO 42010] ISO/IEC/IEEE 42010:2011, Systems and software engineering — Architecture description, online: https://www.iso-architecture.org/ieee-1471/
- [ISO 25010] ISO/IEC DIS 25010(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:dis:ed-2:v1:en

- [iSAQB References] Gernot Starke et. al. Annotated collection of Software Architecture References, for Foundation and Advanced Level Curricula. Freely available https://leanpub.com/isaqbreferences.
- [Keeling 2017] Michael Keeling. Design It!: From Programmer to Software Architect. Pragmatic Programmer.
- [Lange 2021] Kenneth Lange: The Functional Core, Imperative Shell Pattern, online: https://www.kennethlange.com/functional-core-imperative-shell/
- [Lilienthal 2019] Carola Lilienthal: Langlebige Softwarearchitekuren. 3. Auflage, dpunkt Verlag 2019.
- [Lilienthal 2019] Carola Lilienthal: Sustainable Software Architecture: Analyze and Reduce Technical Debt. dpunkt Verlag 2019.
- [Lorz+2021] Alexander Lorz, Gernot Starke: Software Architecture Foundation, CPSA Foundation® Exam Preparation. Van Haaren Publishing, 2021. Alexander Lorz, Gernot Starke
- [Martin 2003] Robert Martin: Agile Software Development. Principles, Patterns, and Practices. Prentice Hall, 2003.
- [Martin 2017] Robert Martin. Clean Architecture: A craftsman's guide to software structure and design. Pearson, 2017.
- [Miller et. al] Heather Miller, Nat Dempkowski, James Larisch, Christopher Meiklejohn: Distributed Programming (to appear, but content-complete) https://github.com/heathermiller/dist-prog-book.
- [Newman 2015] Sam Newman. Building Microservices: Designing Fine-Grained Systems. O'Reilly. 2015.
- [Nygard 2011] Michael Nygard: Documenting Architecture Decision. https://cognitect.com/blog/2011/11/15/documenting-architecture-decisions. See also https://adr.github.io/
- [Nygard 2022] Michael Nygard: CUPID - for joyful coding. See https://dannorth.net/2022/02/10/cupid-for-joyful-coding/.
- [Pethuru 2017] Raj Pethuru et. al: Architectural Patterns. Packt 2017.
- [Starke 2020] Gernot Starke: Effektive Softwarearchitekturen - Ein praktischer Leitfaden (in German). 9. Auflage, Carl Hanser Verlag 2020. Website: https://esabuch.de
- [Eilebrecht+2019] Karl Eilebrecht, Gernot Starke: Patterns kompakt: Entwurfsmuster für effektive Software-Entwicklung (in German). 5th Edition Springer Verlag 2019.
- [UML] The UML reading room, collection of UML resources https://www.omg.org/technology/readingroom/UML.htm. See also https://www.uml-diagrams.org/.
- [vanSteen+Tanenbaum] Andrew Tanenbaum, Maarten van Steen: Distributed Systems, Principles and Paradigms. https://www.distributed-systems.net/.
- [Yorgey 2012] Brent A. Yorgey, Monoids: Theme and Variations. Proceedings of the 2012 Haskell Symposium, September 2012 https://doi.org/10.1145/2364506.2364520
- [Zörner 2021] Stefan Zörner: Softwarearchitekturen dokumentieren und kommunizieren. 3. Auflage, Carl Hanser Verlag, 2021.