Curriculum für

Certified Professional for Software Architecture (CPSA)®

Foundation Level

2023.1.RC-2-DE-20221213

Inhaltsverzeichnis

Die Abkürzung "e. V." ist Teil des offiziellen Namens des iSAQB und steht für "eingetragener Verein", der seinen Status als juristische Person nach deutschem Recht beschreibt. Der Einfachheit halber wird iSAQB e. V. im Folgenden ohne die Verwendung dieser Abkürzung als iSAQB bezeichnet.

Verzeichnis der Lernziele

- LZ 4-2: Qualitätsanforderungen an Softwarearchitekturen klären (R1)
- LZ 4-3: Softwarearchitekturen qualitativ analysieren (R2-R3)
- LZ 4-4: Softwarearchitekturen quantitativ bewerten (R2)
- LZ 5-1: Bezug von Anforderungen und Randbedingungen zu Lösung erfassen (R3)
- LZ 5-2: Technische Umsetzung einer Lösung nachvollziehen (R3)

Einleitung

Was vermittelt eine Foundation-Level-Schulung?

Lizenzierte Schulungen zum *Certified Professional for Software Architecture – Foundation Level* (CPSA-F) vermitteln grundlegende Kenntnisse und Fertigkeiten für den Entwurf einer angemessenen Softwarearchitektur für kleine und mittlere IT-Systeme. Die Teilnehmenden erweitern und vertiefen ihre bestehenden Erfahrungen und Fähigkeiten in der Softwareentwicklung, um relevante Vorgehensweisen, Methoden und Prinzipien für die Entwicklung von Softwarearchitekturen. Durch das Gelernte können sie auf Grundlage angemessen detaillierter Anforderungen und Randbedingungen eine adäquate Softwarearchitektur entwerfen, kommunizieren, analysieren, bewerten und weiterentwickeln. Schulungen zum CPSA-F vermitteln Grundlagenwissen unabhängig von spezifischen Entwurfsmethoden, Vorgehensmodellen, Programmiersprachen oder Werkzeugen. Dadurch können die Teilnehmenden ihre erworbene Fertigkeiten auf ein breites Spektrum von Einsatzfällen anwenden.

Im Mittelpunkt steht der Erwerb folgender Fähigkeiten:

- mit anderen Beteiligten aus den Bereichen Anforderungsmanagement, Projektmanagement, Entwicklung und Test wesentliche Architekturentscheidungen abzustimmen
- die wesentlichen Schritte beim Entwurf von Softwarearchitekturen zu verstehen sowie für kleine und mittlere Systeme selbstständig durchzuführen
- Softwarearchitekturen auf Basis von Sichten, Architekturmustern und technischen Konzepten zu dokumentieren und zu kommunizieren.

Weiterhin behandeln CPSA-F-Schulungen:

- den Begriff und die Bedeutung von Softwarearchitektur
- die Aufgaben und Verantwortung von Softwarearchitekten
- die Rolle von Softwarearchitekt:innen in Entwicklungsvorhaben
- State-of-the-Art-Methoden und -Praktiken zur Entwicklung von Softwarearchitekturen.

Abgrenzung

Dieser Lehrplan reflektiert den aus heutiger Sicht des iSAQB e.V. notwendigen und sinnvollen Inhalt zur Erreichung der Lernziele des CPSA-F. Er stellt keine vollständige Beschreibung des Wissensgebiets „Softwarearchitektur“ dar.

Folgende Themen oder Konzepte sind **nicht Bestandteil des CPSA-F**:

- konkrete Implementierungstechnologien, -frameworks oder -bibliotheken
- Programmierung oder Programmiersprachen
- Spezifische Vorgehensmodelle
- Grundlagen oder Notationen der Modellierung (wie etwa UML)
- Systemanalyse und Requirements Engineering (siehe dazu das Ausbildungs- und Zertifizierungsprogramm des IREB e. V., https://ireb.org, International Requirements Engineering Board)
- Test (siehe dazu das Ausbildungs- und Zertifizierungsprogramm des ISTQB e. V., https://istqb.org, International Software Testing Qualification Board)
- Projekt- oder Produktmanagement
- Einführung in konkrete Werkzeuge.

Ziel des Trainings ist es, die Grundlagen für den Erwerb der für den jeweiligen Einsatzfall notwendigen weiterführenden Kenntnisse und Fertigkeiten zu vermitteln.

Voraussetzungen

Der iSAQB e. V. kann in Zertifizierungsprüfungen die hier genannten Voraussetzungen durch entsprechende Fragen prüfen.

Teilnehmende sollten die im Nachfolgenden genannten Kenntnisse und/oder Erfahrung mitbringen. Insbesondere bilden substanzielle praktischen Erfahrungen aus der Softwareentwicklung im Team eine wichtige Voraussetzung zum Verständnis des vermittelten Lernstoffes und für eine erfolgreiche Zertifizierung.

- mehr als 18 Monate praktische Erfahrung in arbeitsteiliger Softwareentwicklung (d.h. in Teams), erworben durch Programmierung unterschiedlicher Systeme außerhalb der Ausbildung
- Kenntnisse und praktische Erfahrung in mindestens einer höheren Programmiersprache, insbesondere:
  - Konzepte der
    - Modularisierung (Pakete, Namensräume)
    - Parameterübergabe (*Call-by-Value, Call-by-Reference*)
    - Gültigkeit (*scope*), beispielsweise von Typ- oder Variablendeklaration und -definition
  - Grundlagen von Typsystemen (statische und dynamische Typisierung, generische Datentypen)
  - Fehler- und Ausnahmebehandlung in Software
  - Mögliche Probleme von globalem Zustand und globalen Variablen
- Grundlegende Kenntnisse von:
  - Modellierung und Abstraktion
  - Algorithmen und Datenstrukturen (etwa Listen, Bäume, HashTable, Dictionary/Map)
  - UML (Klassen-, Paket-, Komponenten- und Sequenzdiagramme) und deren Bezug zum Quellcode
  - Vorgehen beim Testen von Software (z.B. Unit- und Akzeptanztests)

Hilfreich für das Verständnis einiger Konzepte sind darüber hinaus:

- Grundbegriffe bzw. Unterschiede von imperativer, deklarativer, objektorientierter und funktionaler Programmierung
- praktische Erfahrung in
  - einer höheren Programmiersprache
  - Konzeption und Implementierung verteilt ablaufender Anwendungen, wie etwa Client/Server-Systeme oder Web-Anwendungen
  - technischer Dokumentation, insbesondere in der Dokumentation von Quellcode, Systementwürfen oder technischen Konzepten

Struktur, Dauer, Didaktik

Die in den nachfolgenden Kapiteln des Lehrplans genannten Zeiten sind lediglich Empfehlungen. Die Dauer einer Schulung sollte mindestens 3 Tage betragen, kann aber durchaus länger sein. Anbieter können sich durch Dauer, Didaktik, Art und Aufbau der Übungen sowie der detaillierten Kursgliederung voneinander unterscheiden. Insbesondere die Art (fachliche und technische Domänen) der Beispiele und Übungen können die jeweiligen Schulungsanbieter individuell festlegen.

Inhalt	Empfohlene Dauer (min)
1. Grundlagen	120
2. Entwurf und Entwicklung	420
3. Beschreibung und Kommunikation	240
4. Architektur und Qualität	120
5. Beispiele	90
Summe	990

Lernziele und Prüfungsrelevanz

Die Kapitel des Lehrplans sind anhand von priorisierten Lernzielen gegliedert. Die Prüfungsrelevanz dieser Lernziele beziehungsweise deren Unterpunkte ist beim jeweiligen Lernziel ausdrücklich gekennzeichnet (durch Angabe der Kennzeichen R1, R2 oder R3, siehe nachstehende Tabelle).

Jedes Lernziel beschreibt die zu vermittelnden Inhalte inklusive ihrer Kernbegriffe und -konzepte. Bezüglich der Prüfungsrelevanz verwendet der Lehrplan folgende Kategorien:

ID	Lernziel-Kategorie	Bedeutung	Relevanz für Prüfung
R1	Können	Diese Inhalte sollen die Teilnehmenden nach der Schulung selbstständig anwenden können. Innerhalb der Schulung werden diese Inhalte durch Übungen und Diskussionen abgedeckt.	Inhalte **werden** geprüft.
R2	Verstehen	Diese Inhalte sollen die Teilnehmenden grundsätzlich verstehen. Sie werden in Schulungen i. d. R. nicht durch Übungen vertieft.	Inhalte **können** geprüft werden.
R3	Kennen	Diese Inhalte (Begriffe, Konzepte, Methoden, Praktiken oder Ähnliches) können das Verständnis unterstützen oder das Thema motivieren. Sie werden in Schulungen bei Bedarf thematisiert.	Inhalte **werden nicht** geprüft.

Bei Bedarf enthalten die Lernziele Verweise auf weiterführende Literatur, Standards oder andere Quellen. Die Abschnitte "Begriffe und Konzepte" zu Beginn jedes Kapitels zeigen Worte, die mit dem Inhalt des Kapitels in Verbindung stehen und z. T. auch in den Lernzielen verwendet werden.

Aktuelle Version sowie öffentliches Repository

Den aktuellen Stand dieses Dokumentes finden Sie auf der offiziellen Download-Seite unter https://isaqb-org.github.io/.

Das Dokument wird in einem öffentlichen Repository unter https://github.com/isaqb-org/curriculum-foundation gepflegt, alle Änderungen sind dort sichtbar.

Bitte melden Sie eventuelle Probleme in unserem öffentlichen Issue Tracker bei https://github.com/isaqb-org/curriculum-foundation/issues.

1. Grundbegriffe

Dauer: 120 Min.	Übungszeit: Keine

Wesentliche Begriffe

Softwarearchitektur; Architekturdomänen; Struktur; Bausteine; Komponenten; Schnittstellen; Beziehungen; Querschnittskonzepte; Nutzen von Softwarearchitektur; Softwarearchitekt:innen und deren Verantwortlichkeiten; Rolle; Aufgaben und benötigte Fähigkeiten; Stakeholder und deren Anliegen; Anforderungen; Randbedingungen; Einflussfaktoren; Typen von IT-Systemen (eingebettete Systeme; Echtzeitsysteme; Informationssysteme etc.)

Lernziele

LZ 1-1: Definitionen von Softwarearchitektur diskutieren (R1)

Softwarearchitekt:innen kennen mehrere Definitionen von Softwarearchitektur (u. a. ISO 42010/IEEE 1471, SEI, Booch etc.) und können deren Gemeinsamkeiten benennen:

- Komponenten/Bausteine mit Schnittstellen und Beziehungen
- Bausteine als allgemeiner Begriff, Komponenten als eine spezielle Ausprägung davon
- Strukturen, Querschnittskonzepte, Prinzipien
- Architekturentscheidungen mit systemweiten oder den gesamten Lebenszyklus betreffenden Konsequenzen

LZ 1-2: Nutzen und Ziele von Softwarearchitektur verstehen und erläutern (R1)

Softwarearchitekt:innen können folgenden Nutzen und wesentlichen Ziele von Softwarearchitektur begründen:

- Entwurf, Implementierung, Pflege und Betrieb von Systemen zu unterstützen
- funktionale Anforderungen zu erreichen bzw. deren Erfüllbarkeit sicherzustellen
- Anforderungen wie Zuverlässigkeit, Wartbarkeit, Änderbarkeit, Sicherheit, Energieeffizienz zu erreichen
- Verständnis für Strukturen und Konzepte des Systems zu vermitteln, bezogen auf sämtliche relevanten Stakeholder
- systematisch Komplexität zu reduzieren
- architekturrelevante Richtlinien für Implementierung und Betrieb zu spezifizieren

LZ 1-3: Softwarearchitektur in Software-Lebenszyklus einordnen (R2)

Softwarearchitekt:innen können Ihre Aufgaben und Ergebnisse in den gesamten Lebenszyklus von IT-Systemen einordnen. Sie können:

- Konsequenzen von Änderungen bei Anforderungen, Technologien oder der Systemumgebung im Bezug auf die Softwarearchitektur erkennen
- inhaltliche Zusammenhänge zwischen IT-Systemen und den unterstützten Geschäfts- und Betriebsprozessen aufzeigen

LZ 1-4: Aufgaben und Verantwortung von Softwarearchitekt:innen verstehen (R1)

Softwarearchitekt:innen tragen die Verantwortung für die Erreichung der Anforderungen und die Entwicklung der Architektur der Lösung. Sie müssen diese Verantwortung, abhängig vom jeweiligen Prozess- oder Vorgehensmodell, mit der Gesamtverantwortung der Projektleitung oder anderen Rollen koordinieren.

Aufgaben und Verantwortung von Softwarearchitekt:innen:

- Anforderungen und Randbedingungen klären, hinterfragen und bei Bedarf verfeinern, insbesondere *Required Features und__Required Constraints*
- Strukturentscheidungen hinsichtlich Systemzerlegung und Bausteinstruktur treffen, dabei Abhängigkeiten und Schnittstellen zwischen den Bausteinen festlegen
- Querschnittskonzepte entscheiden (beispielsweise Persistenz, Kommunikation, GUI) und bei Bedarf umsetzen
- Softwarearchitektur auf Basis von Sichten, Architekturmustern sowie technischen und Querschnittskonzepten kommunizieren und dokumentieren
- Umsetzung und Implementierung der Architektur begleiten, Rückmeldungen der beteiligten Stakeholder bei Bedarf in die Architektur einarbeiten, Konsistenz von Quellcode und Softwarearchitektur prüfen und sicherstellen
- Softwarearchitektur analysieren und bewerten, insbesondere hinsichtlich Risiken bezüglich der Erreichung von Anforderungen. Siehe LZ 4-3: Softwarearchitekturen qualitativ analysieren (R2-R3) und LZ 4-4: Softwarearchitekturen quantitativ bewerten (R2).
- Die Konsequenzen von Architekturentscheidungen erkennen, aufzeigen und gegenüber anderen Stakeholdern argumentieren

Sie sollen selbstständig die Notwendigkeit von Iterationen bei allen Aufgaben erkennen und Möglichkeiten für entsprechende Rückmeldung aufzeigen.

LZ 1-5: Rolle von Softwarearchitekt:innen in Beziehung zu anderen Stakeholdern setzen (R1)

Softwarearchitekt:innen können ihre Rolle erklären. Sie sollten ihren Beitrag zur Systementwicklung in Verbindung mit anderen Stakeholdern und Organisationseinheiten kontextspezifisch ausgestalten, insbesondere zu:

- Produktmanagement, Product-Owner
- Projektleitung und -management
- Anforderungsanalytiker:innen (System-/Businessanalyse, Anforderungsmanagement, Fachbereich)
- Entwicklung
- Qualitätssicherung und Test
- IT-Betrieb (Produktion, Rechenzentren), zutreffend primär für Informationssysteme
- Hardwareentwicklung
- Unternehmensarchitektur, Architekturboard.

LZ 1-6: Zusammenhang zwischen Entwicklungsvorgehen und Softwarearchitektur erläutern können (R2)

- Softwarearchitekt:innen können den Einfluss von iterativem Vorgehen auf Architekturentscheidungen erläutern (hinsichtlich Risiken und Prognostizierbarkeit).
- Sie müssen aufgrund inhärenter Unsicherheit oftmals iterativ arbeiten und entscheiden. Dabei müssen sie bei anderen Stakeholdern systematisch Rückmeldung einholen.

LZ 1-7: Kurz- und langfristige Ziele differenzieren (R2)

Softwarearchitekt:innen können potenzielle Zielkonflikte zwischen kurz- und langfristigen Zielen erklären, um eine für alle Beteiligten tragfähige Lösung zu erarbeiten

LZ 1-8: Explizite von impliziten Aussagen unterscheiden (R1)

Softwarearchitekt:innen:

- können Annahmen oder Voraussetzungen explizit darstellen und dadurch implizite Annahmen vermeiden
- wissen, dass implizite Annahmen potenzielle Missverständnisse zwischen beteiligten Stakeholdern bewirken
- können implizit formulieren, wenn es im gegebenen Kontext angemessen ist

LZ 1-9: Zuständigkeit von Softwarearchitekt:innen in organisatorischen Kontext einordnen (R3)

Der Fokus des iSAQB CPSA-Foundation Level liegt auf Strukturen und Konzepten einzelner Softwaresysteme.

Darüber hinaus kennen Softwarearchitekt:innen weitere Architekturdomänen, beispielsweise:

- Unternehmens-IT-Architektur (*Enterprise IT Architecture*): Struktur von Anwendungslandschaften
- Geschäfts- bzw. Prozessarchitektur (*Business and Process Architecture*): Struktur von u.a. Geschäftsprozessen
- Informationsarchitektur: systemübergreifende Struktur und Nutzung von Information und Daten
- Infrastruktur- bzw. Technologiearchitektur: Struktur der technischen Infrastruktur, Hardware, Netze etc.
- Hardware- oder Prozessorarchitektur (für hardwarenahe Systeme)

Diese Architekturdomänen sind nicht inhaltlicher Fokus vom CPSA-F.

LZ 1-10: Typen von IT-Systemen unterscheiden (R3)

Softwarearchitekt:innen kennen unterschiedliche Typen von IT-Systemen, beispielsweise:

- Informationssysteme
- Decision-Support, Data-Warehouse oder Business-Intelligence Systeme
- Mobile Systeme
- *Cloud-native* Systeme (siehe [Cloud-Native])

- Batchprozesse oder -systeme
- Systeme, die auf maschinellem Lernen oder künstlicher Intelligenz basieren
- hardwarenahe Systeme; hier verstehen sie die Notwendigkeit des Hardware-/Software-Co-Designs (zeitliche und inhaltliche Abhängigkeiten von Hard- und Softwareentwicklung).

LZ 1-11: Herausforderungen verteilter Systeme (R3)

Softwarearchitekt:innen können:

- die Verteilung in einer gegebenen Software-Architektur identifizieren
- Konsistenzkriterien für ein gegebenes fachliches Problem analysieren
- Kausalität von Ereignissen in einem verteilten System erklären

Softwarearchitekt:innen wissen:

- dass Kommunikation in einem verteilten System fehlschlagen kann
- dass es bei verteilten Systemen Einschränkungen hinsichtlich der Konsistenz in Datenbanken gibt
- was das "Split-Brain"-Problem ist und warum es schwierig zu lösen ist
- dass es unmöglich ist, die exakte zeitliche Reihenfolge der Ereignisse in einem verteilten System zu bestimmen

Referenzen

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

2. Entwurf und Entwicklung von Softwarearchitekturen

Dauer: 330 Min.	Übungszeit: 90 Min.

Wesentliche Begriffe

Entwurf; Vorgehen beim Entwurf; Entwurfsentscheidung; Sichten; Schnittstellen; technische Konzepte und Querschnittskonzepte; Architekturmuster; Entwurfsmuster; Mustersprachen; Entwurfsprinzipien; Abhängigkeit; Kopplung; Kohäsion; Top-down- und Bottom-up-Vorgehen; modellbasierter Entwurf; iterativer/inkrementeller Entwurf; Domain-Driven Design

Lernziele

LZ 2-1: Vorgehen und Heuristiken zur Architekturentwicklung auswählen und anwenden können (R1,R3)

Softwarearchitekt:innen können grundlegende Vorgehensweisen der Architekturentwicklung benennen, erklären und anwenden, beispielsweise:

- Top-down- und Bottom-up-Vorgehen beim Entwurf (R1)
- Sichtenbasierte Architekturentwicklung (R1)
- iterativer und inkrementeller Entwurf (R1)
  - Notwendigkeit von Iterationen, insbesondere bei unter Unsicherheit getroffenen Entscheidungen (R1)
  - Notwendigkeit von Rückmeldungen zu Entwurfsentscheidungen (R1)
- Domain-Driven Design, siehe [Evans 2004] (R3)
- Evolutionäre Architektur, siehe [Ford 2017] (R3)
- Globale Analyse, siehe [Hofmeister et. al 1999] (R3)
- Modellgetriebene Architektur (R3)

LZ 2-2: Softwarearchitekturen entwerfen (R1)

Softwarearchitekt:innen können:

- Softwarearchitekturen auf Basis bekannter funktionaler und Qualitätsanforderungen für nicht sicherheits- oder unternehmenskritische Softwaresysteme entwerfen und angemessen kommunizieren und dokumentieren
- Strukturentscheidungen hinsichtlich Systemzerlegung und Bausteinstruktur treffen, dabei Abhängigkeiten zwischen Bausteinen festlegen
- gegenseitige Abhängigkeiten und Abwägungen bezüglich Entwurfsentscheidungen erkennen und begründen
- Begriffe *Blackbox* und *Whitebox* erklären und zielgerichtet anwenden
- schrittweise Verfeinerung und Spezifikation von Bausteinen durchführen
- Architektursichten entwerfen, insbesondere Baustein-, Laufzeit- und Verteilungssicht

- die aus diesen Entscheidungen resultierenden Konsequenzen auf den Quellcode erklären
- fachliche und technische Bestandteile in Architekturen trennen und diese Trennung begründen
- domänenspezifische und technische Bestandteile in Architekturen trennen und diese Trennung begründen
- Risiken von Entwurfsentscheidungen identifizieren.

LZ 2-3: Anforderungen klären und berücksichtigen können (R1-R3)

Softwarearchitekt:innen können Anforderungen (inklusive Randbedingungen als Einschränkungen der Entwurfsfreiheit) klären und berücksichtigen. Sie verstehen, dass ihre Entscheidungen zu weiteren Anforderungen (inklusive Randbedingungen) an das zu entwerfende System, seine Architektur oder den Entwicklungsprozess führen können.

Sie erkennen und berücksichtigen den Einfluss von:

- produktbezogenen Anforderungen und Trends wie (R1)
  - funktionale Anforderungen
  - Qualitätsanforderungen
  - technologische, organisatorische und regulatorische Randbedingungen.
- Technologische Randbedingungen wie
  - bestehende oder geplante Hardware- und Software-Infrastruktur (R1)
  - technologische Beschränkungen für Datenstrukturen und Schnittstellen (R2)
  - Referenzarchitekturen, Bibliotheken, Komponenten und Frameworks (R1)
  - Programmiersprachen (R2)
- Organisatorischen Randbedingungen wie
  - Organisationsstruktur von Entwicklungsteams und Auftraggebenden (R1), insbesondere das Gesetz von Conway (R2)
  - Unternehmens- und Teamkultur (R3)
  - Partnerschaften und Kooperationen (R2)
  - Normen, Richtlinien und Prozessmodelle (z.B. Genehmigungs- und Freigabeprozesse) (R2)
  - Verfügbarkeit von Ressourcen wie Budget, Zeit und Personal (R1)
  - Verfügbarkeit, Qualifikation und Engagement von Mitarbeitenden (R1)
- Regulatorischen Randbedingungen wie (R2)
  - lokale und internationale rechtliche Einschränkungen
  - Vertrags- und Haftungsfragen
  - Datenschutzgesetze und Gesetze zum Schutz der Privatsphäre
  - Fragen der Einhaltung oder Verpflichtungen zur Beweislast
- Trends wie (R3)
  - Markttrends
  - Technologietrends (z.B. Blockchain, Microservices)

- Methodik-Trends (z.B. agil)
- (potenzielle) Auswirkungen weiterer Stakeholderinteressen und vorgegebener oder extern festgelegter Designentscheidungen

LZ 2-4: Querschnittskonzepte entwerfen und umsetzen (R1)

Softwarearchitekt:innen können:

- die Bedeutung von Querschnittskonzepten erklären
- Querschnittskonzepte entscheiden und entwerfen, unter anderem Persistenz, Kommunikation, GUI, Fehlerbehandlung, Nebenläufigkeit, Energieeffizienz
- mögliche wechselseitige Abhängigkeiten dieser Entscheidungen erkennen und beurteilen.

Softwarearchitekt:innen wissen, dass solche Querschnittskonzepte systemübergreifend wiederverwendbar sein können.

LZ 2-5: Wichtige Lösungsmuster beschreiben, erklären und angemessen anwenden (R1, R3)

Softwarearchitekt:innen kennen verschiedene Architekturmuster (siehe unten) und können sie gegebenenfalls anwenden.

Sie wissen (R3):

- dass Muster ein Weg sind, bestimmte Qualitäten für gegebene Probleme und Anforderungen innerhalb gegebener Kontexte zu erreichen
- dass es verschiedene Kategorien von Mustern gibt
- zusätzliche Quellen für Muster, die sich auf ihre spezifische technische oder Anwendungsdomäne beziehen

Softwarearchitekt:innen können die folgenden Muster erklären und Beispiele dafür liefern (R1):

- Schichten (Layers):
  - Abstraktionsschichten (Abstraction layers) verbergen Details, Beispiel: ISO/OSI-Netzwerkschichten oder "Hardware-Abstraktionsschicht". Siehe https://en.wikipedia.org/wiki/Hardware_abstraction.
  - Eine andere Interpretation sind Schichten zur (physischen) Trennung von Funktionalität oder Verantwortung, siehe https://en.wikipedia.org/wiki/Multitier_architecture.
- Pipes-and-Filter: Repräsentativ für Datenflussmuster, die die schrittweise Verarbeitung in eine Reihe von Verarbeitungsaktivitäten ("Filter") und zugehörige Transport/Puffer ("Pipes") separieren.
- Microservices teilen Anwendungen in separate ausführbare Dienste auf, die über Netzwerk (remote) kommunizieren. [Newman 2015]
- Dependency Injection als eine mögliche Lösung für das Dependency-Inversion-Prinzip

Softwarearchitekt:innen können einige der folgendene Muster erklären, ihre Relevanz für konkrete Systeme erläutern und Beispiele dafür liefern (R3):

- Blackboard: Behandlung von Problemen, die nicht durch deterministische Algorithmen lösbar sind, sondern vielfältiges Wissen erfordern.

- Broker: verantwortlich für die Koordination der Kommunikation zwischen Anbieter(n) und Verbraucher(n), angewandt in verteilten Systemen. Verantwortlich für die Weiterleitung von Anfragen und/oder die Übermittlung von Ergebnissen, Fehlern und Ausnahmen.
- Kombinator (Synonym: Closure of Operations), für Domänenobjekte vom Typ T, suchen Sie nach Operationen sowohl mit Input- als auch Output-Typ T. Siehe [Yorgey 2012]
- CQRS (Command-Query-Responsibility-Segregation): Trennung von Lese- und Schreibvorgängen in Informationssystemen. Erfordert Einblicke in konkrete Datenbank-/Persistenztechnologie, um die unterschiedlichen Eigenschaften und Anforderungen von "Lese-" und "Schreib"-Operationen zu verstehen.
- Event-Sourcing: Behandlung von Datenoperationen durch eine Abfolge von Ereignissen (Events), von denen jedes in einem Append-only Speicher aufgezeichnet wird.
- Interpreter: repräsentieren Domänenobjekt oder DSL als Syntax, bieten eine Funktion, die eine semantische Interpretation des Domänenobjekts getrennt vom Domänenobjekt selbst implementiert.
- Integrations- oder Messaging-Patterns (z.B. aus Hohpe+2004])
- Die MVC- (Model View Controller), MVVM- (Model View ViewModel), MVU- (Model View Update), PAC- (Presentation Abstraction Control) Musterfamilie, die die externe Repräsentation (Ansicht) von Daten von Operationen Diensten und deren Koordination trennt.
- Schnittstellenmuster wie Adapter, Fassade, Proxy. Solche Muster helfen bei der Integration von Subsystemen und/oder bei der Vereinfachung von Abhängigkeiten. Architekt:innen sollten wissen, dass diese Muster unabhängig von (Objekt-)Technologie verwendet werden können.
  - Adapter: Entkopplung von Konsument und Provider - wenn die Schnittstelle des Providers nicht genau mit der des Konsumenten übereinstimmt.
  - Fassade: vereinfacht die Verwendung eines Providers für den/die Consumer durch vereinfachten Zugriff.
  - Proxy: Ein Vermittler/Stellvertreter zwischen Consumer und Provider, der beispielsweise die zeitliche Entkopplung, das Caching von Ergebnissen oder die Zugriffskontrolle auf den Provider ermöglicht.
- Observer (Beobachter): ein Produzent von Werten benachrichtigt eine zentrale Vermittlungsstelle, bei der sich Interessenten (Consumer) registrieren können, um über Änderungen benachrichtigt zu werden.
- Plug-In: erweitert das Verhalten einer Komponente.
- Ports&Adapters (syn. Onion-Architecture, Hexagonale Architektur, Clean-Architecture): konzentrieren die Domänenlogik im Zentrum des Systems, und besitzen lediglich an den Rändern Verbindungen zur Außenwelt (Datenbank, UI). Abhängigkeiten von außen nach innen (Outside-In), niemals von innen nach außen (Inside-Out). [Lange 2021] [Martin 2017]
- Remote Procedure Call: eine Funktion oder einen Algorithmus in einem anderen Adressraum ausführen lassen.
- SOA (Service-orientierte Architektur): Ein Ansatz zur Bereitstellung abstrakter Dienste statt konkreter Implementierungen für die Benutzer des Systems, um die Wiederverwendung von Diensten über Abteilungen und zwischen Unternehmen zu fördern.
- Template und Strategy: spezifische Algorithmen durch Kapselung flexibel machen.
- Visitor (Besucher): Traversierung von Datenstrukturen von spezifischer Verarbeitung trennen.

Softwarearchitekt:innen kennen wesentliche Quellen für Architekturmuster, beispielsweise die POSA-

Literatur (z.B. [Buschmann+ 1996]) und PoEAA ([Fowler 2002]) (für Informationssysteme) (R3)

LZ 2-6: Entwurfsprinzipien erläutern und anwenden (R1-R3)

Softwarearchitekt:innen sind in der Lage zu erklären, was Entwurfsprinzipien sind. Sie können deren grundlegende Ziele und deren Anwendung im Hinblick auf Softwarearchitektur skizzieren. (R2)

Softwarearchitekt:innen sind in der Lage:

- die unten aufgeführten Gestaltungsprinzipien zu erläutern und mit Beispielen zu illustrieren
- zu erklären, wie diese Prinzipien angewendet werden sollen
- darzulegen, wie Anforderungen die Anwendung dieser Prinzipien beeinflussen
- die Auswirkungen der Entwurfsprinzipien auf die Implementierung zu erläutern
- Quellcode und Architektur zu analysieren, um zu beurteilen, ob diese Entwurfsprinzipien angewendet wurden oder angewendet werden sollten

Abstraktion (R1)

- im Sinne eines Vorgehens zur Erarbeitung zweckmäßiger Generalisierungen
- als eine Entwurfstechnik, bei dem die Bausteine von Abstraktionen und nicht von Implementierungen abhängen
- Schnittstellen als Abstraktionen

Modularisierung (R1)

- Geheimnisprinzip (Information Hiding) und Kapselung (R1)
- Trennung von Verantwortlichkeiten (Separation of Concerns - SoC) (R1)
- Lose, aber funktionell ausreichende Kopplung (R1) von Bausteinen, siehe Lernziel 2-7
- Hohe Kohäsion (R1)
- SOLID-Prinzipien (R1-R3), soweit sie auf architektonischer Ebene von Relevanz sind:
  - S: Single-Responsibility-Prinzip (R1) und seine Beziehung zu SoC
  - O: Offen/geschlossen-Prinzip (R1)
  - L: Liskov'sches Substitutionsprinzip (R3) als eine Möglichkeit, Konsistenz und konzeptionelle Integrität beim objektorientierten Design zu erreichen
  - I: Interface-Segregation-Prinzip (R2) und seine Beziehung zu Lernziel 2-9 "Schnittstellen entwerfen und festlegen"
  - D: Dependency-Inversion-Prinzip (R1) - Umkehrung von Abhängigkeiten (R1) durch Schnittstellen oder ähnlichen Abstraktionen

Konzeptionelle Integrität (R2)

- bedeutet Einheitlichkeit (Homogenität, Konsistenz) von Lösungen für ähnliche Probleme zu erreichen (R2)
- als ein Mittel, um das Prinzip der geringsten Überraschung zu erreichen (principle of least surprise) (R3)

Einfachheit (R1)

- als Mittel zur Verringerung von Komplexität (R1)
- als Motiv der Prinzipien KISS (R1) und YAGNI (R2)

Erwarte Fehler (R1-R2)

- als Mittel für den Entwurf robuster und widerstandsfähiger Systeme (R1)
- als eine Verallgemeinerung des Robustheitsgrundsatzes (*Postel's law*) (R2)

Weitere Prinzipien (R3)

Softwarearchitekt:innen kennen weitere Prinzipien (etwa CUPID, siehe [Nygard 2022]) und können diese anwenden.

LZ 2-7: Abhängigkeiten von Bausteinen managen (R1)

Softwarearchitekt:innen verstehen Abhängigkeiten und Kopplung zwischen Bausteinen und können diese gezielt einsetzen. Sie:

- kennen und verstehen unterschiedliche Arten der Kopplung von Bausteinen (beispielsweise Kopplung über Benutzung/Delegation, Nachrichten/Ereignisse, Komposition, Erzeugung, Vererbung, zeitliche Kopplung, Kopplung über Daten, Datentypen oder Hardware)
- verstehen, wie Abhängigkeiten die Kopplung vergrößern
- können solche Arten der Kopplung gezielt einsetzen und die Konsequenzen solcher Abhängigkeiten einschätzen
- kennen Möglichkeiten zur Auflösung bzw. Reduktion von Kopplung und können diese anwenden, beispielsweise:
  - Muster (siehe LZ 2-5)
  - Grundlegende Entwurfsprinzipien (siehe LZ 2-6)
  - Externalisierung von Abhängigkeiten, d.h. konkrete Abhängigkeiten erst zur Installations- oder Laufzeit festlegen, etwa durch Anwendung von *Dependency Injection*.

LZ 2-8: Qualitätsanforderungen mit passenden Ansätzen und Techniken erreichen (R1)

Softwarearchitekt:innen kennen und berücksichtigen den starken Einfluss von Qualitätsanforderungen in Architektur- und Entwurfsentscheidungen, beispielsweise für:

- Effizienz, Laufzeitperformance
- Verfügbarkeit
- Wartbarkeit, Modifizierbarkeit, Erweiterbarkeit, Adaptierbarkeit
- Energieeffizienz

Sie können:

- Lösungsmöglichkeiten, *Architectural Tactics*, angemessene Praktiken sowie technische Möglichkeiten zur Erreichung wichtiger Qualitätsanforderungen von Softwaresystemen (unterschiedlich für eingebettete Systeme bzw. Informationssysteme) erklären und anwenden

- mögliche Wechselwirkungen zwischen solchen Lösungsmöglichkeiten sowie die entsprechenden Risiken identifizieren und kommunizieren.

LZ 2-9: Schnittstellen entwerfen und festlegen (R1-R3)

Softwarearchitekt:innen kennen die hohe Bedeutung von Schnittstellen. Sie können Schnittstellen zwischen Architekturbausteinen sowie externe Schnittstellen zwischen dem System und Elementen außerhalb des Systems entwerfen bzw. festlegen.

Sie kennen:

- wünschenswerte Eigenschaften von Schnittstellen und können diese bei ihrer Entwicklung einsetzen:
  - einfach zu erlernen, einfach zu benutzen, einfach zu erweitern
  - schwer zu missbrauchen
  - funktional vollständig aus Sicht der Nutzer:innen oder nutzender Bausteine.
- die Notwendigkeit unterschiedlicher Behandlung interner und externer Schnittstellen
- unterschiedliche Implementierungsansätze von Schnittstellen (R3):
  - ressourcenorientierter Ansatz (REST, REpresentational State Transfer)
  - serviceorientierter Ansatz (wie bei WS-*/SOAP-basierten Webservices).

LZ 2-10: Grundlegende Prinzipien von Software-Deployments kennen (R3)

Softwarearchitekt:innen:

- wissen, dass Software-Deployment der Prozess ist, durch den neue oder aktualisierte Software zur Benutzung bereitgestellt wird
- können grundlegende Konzepte des Deployments von Software benennen und erklären, beispielsweise:
  - Automatisierung von Deployments
  - Wiederholbare Builds
  - Konsistente Umgebungen (z.B. durch Nutzung von unveränderlicher (*immutable*) Infrastruktur)
  - Alles liegt unter Versionskontrolle
  - Releases sind einfach zurückzunehmen

Referenzen

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

3. Beschreibung und Kommunikation von Softwarearchitekturen

Dauer: 180 Min.	Übungszeit: 60 Min.

Wesentliche Begriffe

(Architektur-)Sichten; Strukturen; (technische) Konzepte; Dokumentation; Kommunikation; Beschreibung; zielgruppen- oder stakeholdergerecht; Meta-Strukturen und Templates zur Beschreibung und Kommunikation; Kontextabgrenzung; Bausteine; Bausteinsicht; Laufzeitsicht; Verteilungssicht; Knoten; Kanal; Verteilungsartefakte; Mapping von Bausteinen auf Verteilungsartefakte; Beschreibung von Schnittstellen und Entwurfsentscheidungen; UML; Werkzeuge zur Dokumentation

Lernziele

LZ 3-1: Anforderungen an technische Dokumentation erläutern und berücksichtigen (R1)

Softwarearchitekt:innen kennen die wesentlichen Anforderungen an technische Dokumentation und können diese bei der Dokumentation von Systemen berücksichtigen bzw. erfüllen:

- Verständlichkeit, Korrektheit, Effizienz, Angemessenheit, Wartbarkeit
- Orientierung von Form, Inhalt und Detailgrad an Zielgruppe der Dokumentation

Sie wissen, dass Verständlichkeit technischer Dokumentation nur von deren Zielgruppen beurteilt werden kann.

LZ 3-2: Softwarearchitekturen beschreiben und kommunizieren (R1, R3)

Softwarearchitekt:innen nutzen Dokumentation zur Unterstützung bei Entwurf, Implementierung und Weiterentwicklung (auch genannt *Wartung* oder *Evolution*) von Systemen. (R2)

Softwarearchitekt:innen (R1):

- können Architekturen stakeholdergerecht dokumentieren und kommunizieren und dadurch unterschiedliche Zielgruppen adressieren, z. B. Management, Entwicklungsteams, QS, andere Softwarearchitekt:innen sowie möglicherweise zusätzliche Stakeholder
- können die Beiträge unterschiedlicher Autorengruppen stilistisch und inhaltlich konsolidieren und harmonisieren
- können Maßnahmen entwickeln und umsetzen, die mündliche und schriftliche Kommunikation in Einklang miteinander halten und miteinander angemessen ausbalancieren
- kennen den Nutzen von Template-basierter Dokumentation
- wissen, dass verschiedene Eigenschaften der Dokumentation von Spezifika des Systems, seinen Anforderungen, Risiken, dem Entwicklungsvorgehen, der Organisation oder anderen Faktoren abhängen.

Sie können beispielsweise die folgenden Merkmale von Dokumentation je nach Situation anpassen (R3):

- Umfang und Detaillierungsgrad der benötigten Dokumentation
- das Dokumentationsformat

- die Zugänglichkeit der Dokumentation
- Formalitäten der Dokumentation (z.B. Diagramme, die einem Metamodell entsprechen, oder einfache Zeichnungen)
- formale Überprüfungen und Freigabeprozesse für die Dokumentation

LZ 3-3: Notations-/Modellierungsmittel für Beschreibung von Softwarearchitektur erläutern und anwenden (R2-R3)

Softwarearchitekt:innen kennen mindestens folgende UML-Diagramme (siehe [UML]) zur Notation von Architektursichten:

- Klassen-, Paket-, Komponenten- (jeweils R2) und Kompositionsstrukturdiagramme (R3)
- Verteilungsdiagramme (R2)
- Sequenz- und Aktivitätsdiagramme (R2)
- Zustandsdiagramme (R3)

Softwarearchitekt:innen kennen Alternativen zu UML, beispielsweise (R3)

- ArchiMate, siehe [Archimate]
- für Laufzeitsichten beispielsweise Flussdiagramme, nummerierte Listen oder Business-Process-Modelling-Notation (BPMN).

LZ 3-4: Architektursichten erläutern und anwenden (R1)

Softwarearchitekt:innen können folgende Architektursichten anwenden:

- Kontextsicht (auch genannt Kontextabgrenzung)
- Baustein- oder Komponentensicht (Aufbau des Systems aus Softwarebausteinen)
- Laufzeitsicht (dynamische Sicht, Zusammenwirken der Softwarebausteine zur Laufzeit, Zustandsmodelle)
- Verteilungs-/Deploymentsicht (Hardware und technische Infrastruktur sowie Abbildung von Softwarebausteinen auf diese Infrastruktur)

LZ 3-5: Kontextabgrenzung von Systemen erläutern und anwenden (R1)

Softwarearchitekt:innen können:

- Kontext von Systemen z.B. in Form von Kontextdiagrammen mit Erläuterungen darstellen
- externe Schnittstellen von Systemen in der Kontextabgrenzung darstellen
- fachlichen und technischen Kontext differenzieren.

LZ 3-6: Querschnittskonzepte dokumentieren und kommunizieren (R2)

Softwarearchitekt:innen können typische Querschnittskonzepte (synonym *Prinzipien*, *Aspekte*) adäquat dokumentieren und kommunizieren, z. B. Persistenz, Ablaufsteuerung, UI, Verteilung/Integration, Protokollierung.

LZ 3-7: Schnittstellen beschreiben (R1)

Softwarearchitekt:innen können sowohl interne als auch externe Schnittstellen beschreiben und spezifizieren.

LZ 3-8: Architekturentscheidungen erläutern und dokumentieren (R1-R2)

Softwarearchitekt:innen können:

- Architekturentscheidungen systematisch herbeiführen, begründen, kommunizieren und dokumentieren
- gegenseitige Abhängigkeiten solcher Entscheidungen erkennen, kommunizieren und dokumentieren

Softwarearchitekt:innen kennen Architecture-Decision-Records (ADR, siehe [Nygard 2011]) und können diese zur Dokumentation von Entscheidungen einsetzen (R2).

LZ 3-9: Weitere Hilfsmittel und Werkzeuge zur Dokumentation kennen (R3)

Softwarearchitekt:innen kennen:

- Grundlagen mehrerer publizierter Frameworks zur Beschreibung von Softwarearchitekturen, beispielsweise:
  - ISO/IEEE-42010 (vormals 1471), siehe [ISO 42010]
  - arc42, siehe [arc42]
  - C4, siehe [Brown]
  - FMC, siehe [FMC]
- Ideen und Beispiele von Checklisten für die Erstellung, Dokumentation und Prüfung von Softwarearchitekturen
- mögliche Werkzeuge zur Erstellung und Pflege von Architekturdokumentation

Referenzen

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

4. Softwarearchitektur und Qualität

Dauer: 60 Min.	Übungszeit: 60 Min.

Wesentliche Begriffe

Qualität; Qualitätsmerkmale; DIN/ISO 25010; Qualitätsszenarien; Qualitätsbaum; Kompromisse/Wechselwirkungen von Qualitätseigenschaften; qualitative Architekturanalyse; Metriken und quantitative Analyse

Lernziele

LZ 4-1: Qualitätsmodelle und Qualitätsmerkmale diskutieren (R1)

Softwarearchitekt:innen können:

- den Begriff der Qualität und der Qualitätsmerkmale (angelehnt an [ISO 25010]) erklären
- generische Qualitätsmodelle (wie etwa [ISO 25010]) erklären
- Zusammenhänge und Wechselwirkungen von Qualitätsmerkmalen erläutern, beispielsweise:
  - Konfigurierbarkeit versus Zuverlässigkeit
  - Speicherbedarf versus Leistungseffizienz
  - Sicherheit versus Benutzbarkeit
  - Laufzeitflexibilität versus Wartbarkeit.

LZ 4-2: Qualitätsanforderungen an Softwarearchitekturen klären (R1)

Softwarearchitekt:innen können:

- spezifische Qualitätsanforderungen an die zu entwickelnde Software und deren Architekturen klären und konkret formulieren, beispielsweise in Form von Szenarien und Qualitätsbäumen
- Szenarien und Qualitätsbäume erklären und anwenden.

LZ 4-3: Softwarearchitekturen qualitativ analysieren (R2-R3)

Softwarearchitekt:innen:

- kennen methodische Vorgehensweisen zur qualitativen Analyse von Softwarearchitekturen (R2), beispielsweise nach ATAM (R3)
- können kleinere Systeme qualitativ analysieren (R2)
- wissen, dass zur qualitativen Analyse und Bewertung von Architekturen folgende Informationsquellen helfen können (R2):
  - Qualitätsanforderungen, beispielsweise in Form von Qualitätsbäumen und -szenarien
  - Architekturdokumentation
  - Architektur- und Entwurfsmodelle
  - Quellcode

    - Metriken
    - Sonstige Dokumentationen des Systems, etwa Anforderungs-, Betriebs- oder Testdokumentation.

LZ 4-4: Softwarearchitekturen quantitativ bewerten (R2)

Softwarearchitekt:innen kennen Ansätze zur quantitativen Analyse und Bewertung (Messung) von Software.

Sie wissen, dass:

- quantitative Bewertung helfen kann, kritische Teile innerhalb von Systemen zu identifizieren
- zur Bewertung von Architekturen weitere Informationen hilfreich sein können, etwa:
    - Anforderungs- und Architekturdokumentation
    - Quellcode und diesbezügliche Metriken wie Lines-of-Code, (zyklomatische) Komplexität, ein- und ausgehende Abhängigkeiten
    - bekannte Fehler in Quellcode, insbesondere Fehlercluster
    - Testfälle und Testergebnisse.
- die Verwendung einer Metrik als Zielgröße zu deren Entwertung führen kann (R2), wie z. B. durch Goodharts Gesetz (R3) beschrieben.

Referenzen

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

5. Beispiele für Softwarearchitekturen

Dauer: 90 Min.	Übungszeit: Keine

Dieser Abschnitt ist nicht prüfungsrelevant.

Lernziele

LZ 5-1: Bezug von Anforderungen und Randbedingungen zu Lösung erfassen (R3)

Softwarearchitekt:innen haben an mindestens einem Beispiel den Bezug von Anforderungen und Randbedingungen zu Lösungsentscheidungen erkannt und nachvollzogen.

LZ 5-2: Technische Umsetzung einer Lösung nachvollziehen (R3)

Softwarearchitekt:innen können anhand mindestens eines Beispiels die technische Umsetzung (Implementierung, technische Konzepte, eingesetzte Produkte, Lösungsstrategien) einer Lösung nachvollziehen.

Referenzen

- [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.