SOFTWAREENTWICKLUNG

PROJEKTMANAGEMENT

Inhalt

- Projektmanagement
- Informations- und Hinweispflichten
- Grundlagen von Programmiersprachen
- Funktionale Programmierung
- Objektorientierung
- Asynchrone Programmierung
- Algorithmen
- Software-Engineering
- Softwarearchitektur
- Schnittstellen
- Containerisierung
- App-Entwicklung
- Pseudococde und UML
- Hardware

Übersicht

Projektmanagement

- Was ist ein Projekt?
- Projektphasen Allgemein
- Vorgehensmodelle
- Projektplanung (Klassisch)
- Netzplan und Gantt Chart
- Bedarfsanalyse
- Analyseverfahren
- Designverfahren
- Lasten- und Pflichtenheft
- Funktionale und Nicht-Funktionale Anforderungen
- Spezifikationen
- Angebotsbewertung
- Eigenfertigung versus Fremdfertigung

Was ist ein Projekt?

Verfolgt ein Ziel, zeitlich befristet, relativ innovativ, begrenzte personelle und finanzielle Ressourcen, risikobehaftet, erhebliche Komplexität, erfordert ein Projektmanagement

Qualität = Übereinstimmung mit Anforderungen

SMARTe Ziele (spezifisch, messbar, akzeptiert/attraktiv, realistisch, terminiert)

Projektphasen Allgemein

Vorgehensmodelle

Klassisch

Wasserfallmodell

Iterative Modelle

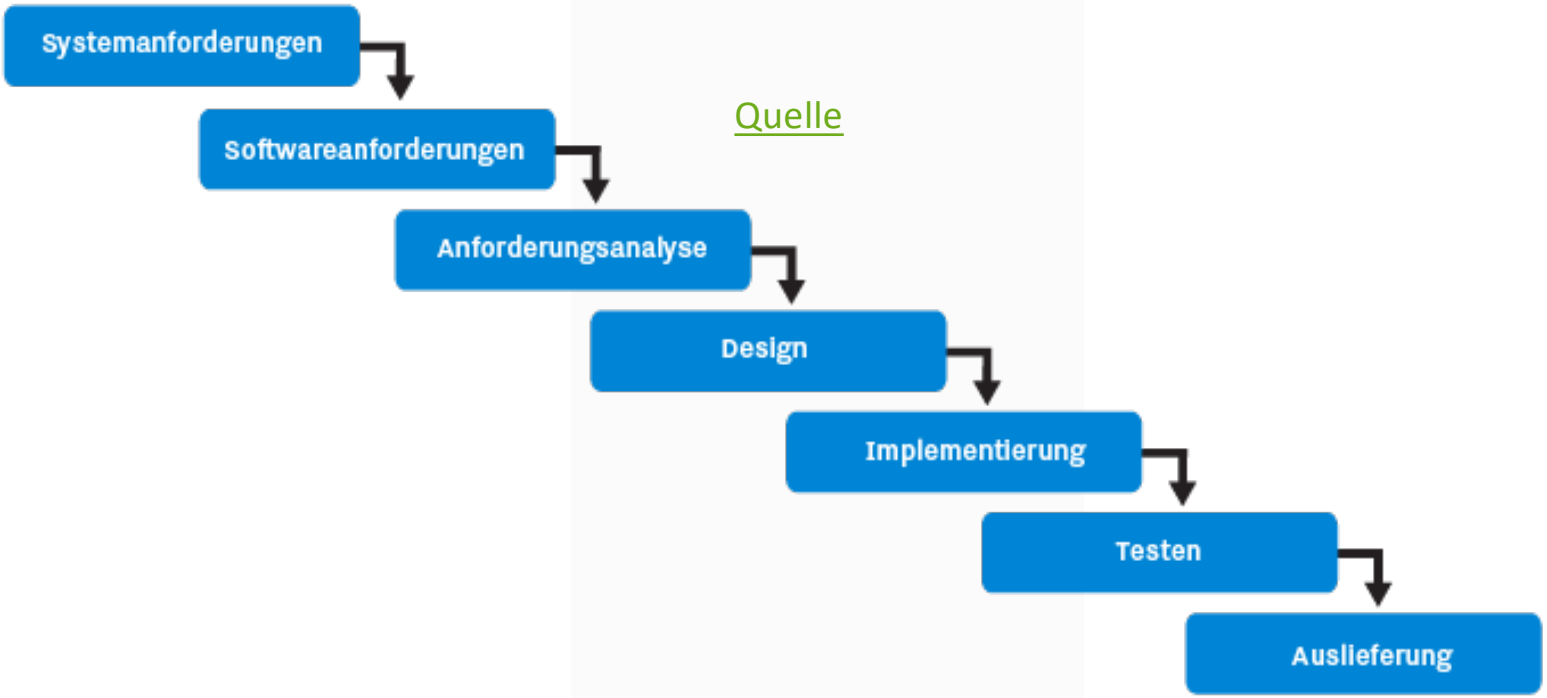
Agil

SCRUM

Kanban

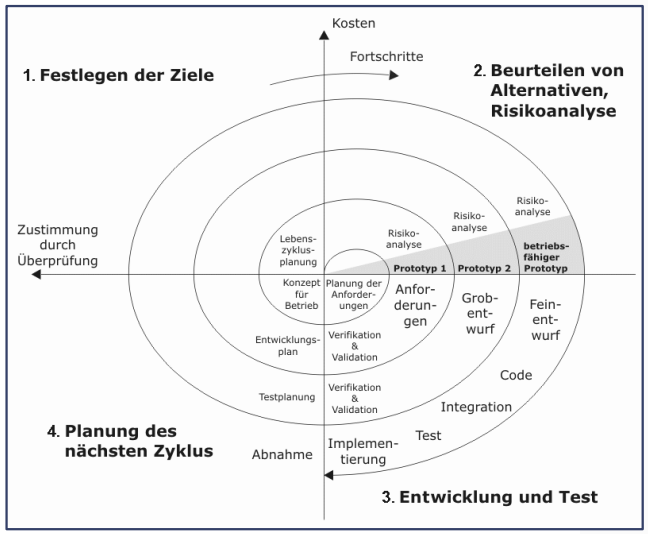
XP

Wasserfallmodell

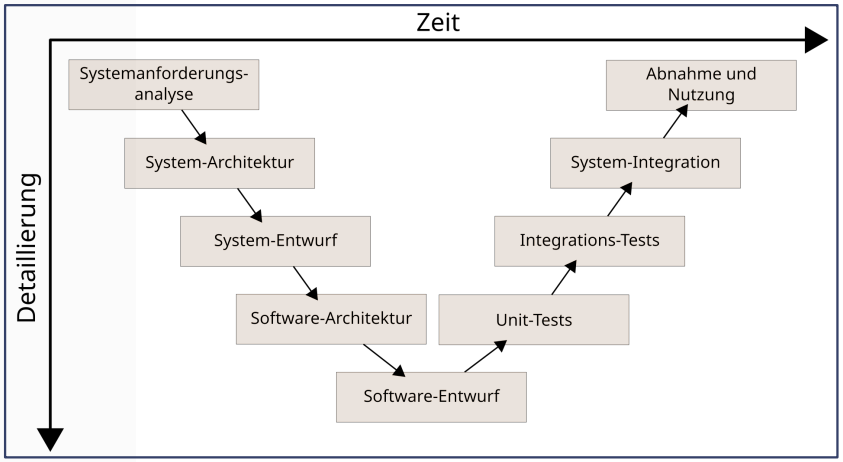


Quelle

Spiralmodell versus V-Modell (XT)



Quelle

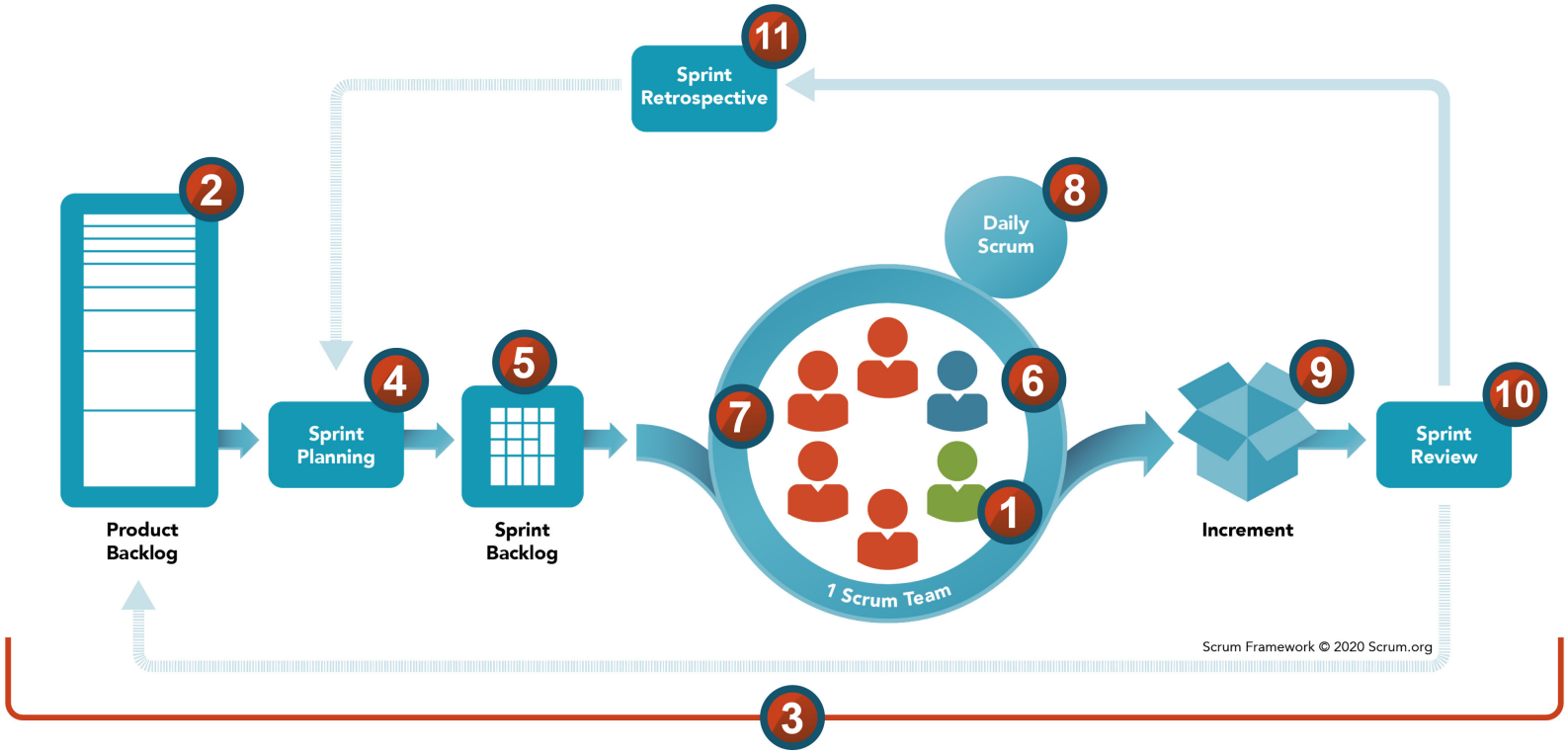


Quelle

- Frühes Erkennen und Minimieren von Risiken
- Anpassung an Änderungen möglich
- Feedback in jeder Iteration

- Klare Struktur und Verantwortlichkeiten
- Starke Qualitätssicherung
- Gut geeignet für Projekte mit stabilen Anforderungen

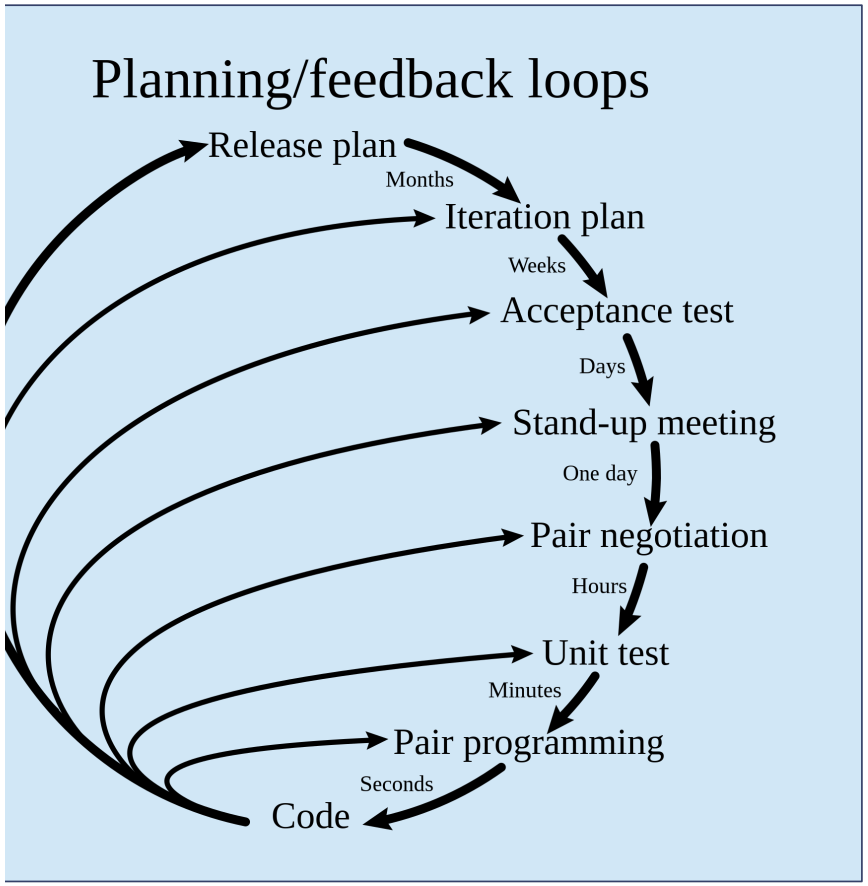
SCRUM



5 Events
- Sprint
- Sprint Planning
- Daily Scrum
- Sprint Review
- Sprint Retrospective

3 Artefakte
- Product Backlog
- Sprint Backlog
- Product Increment

Scrum-Team
- Product Owner
- Developer
- Scrum Master



Extreme Programming (XP)

- Fokus auf hohe Softwarequalität und kurze Entwicklungszyklen
- **Merkmale:**
  - Kurze Iterationen (1–2 Wochen)
  - Test-Driven Development (TDD)
  - Pair Programming
  - Kontinuierliche Integration
  - Refactoring (kontinuierliche Verbesserung des Codes)
  - Kundenbeteiligung im gesamten Prozess
- Vorteile:
  - Hohe Codequalität
  - Schnelle Anpassung an Änderungen
  - Starker Fokus auf Auftraggebende

Kanban

- Visuelles **Workflow-Management-System** aus der Lean-Production (Toyota), angepasst für Softwareentwicklung
- **Merkmale:**
  - Kanban-Board mit Spalten wie „To Do“, „In Progress“, „Done“
  - Work in Progress (WIP) Limits → Begrenzung paralleler Aufgaben
  - Kontinuierlicher Fluss statt fester Iterationen
  - Fokus auf Prozessoptimierung und Engpassbeseitigung
- **Vorteile:**
  - Flexibel, kein Sprint-Zwang
  - Sehr gute Visualisierung des Projektstatus
  - Einfacher Einstieg

Projektplanung (klassisch) 1/2

- Die Projektplanung dient dazu:
  - den kritischer Pfad aufzuzeigen
  - Pufferzeiten zu erkennen
  - eine fristgerechte Terminierung sicher zu stellen
  - Lösungsmöglichkeiten bei Terminproblemen zu finden
  - Meilensteine festzulegen

Projektplanung (klassisch) 2/2

- **Aufgaben der Projektplanung:**
  - Definieren und Festlegen von Arbeitspaketen und Abhängigkeiten
  - Erleichterung der Planung und der Fortschrittskontrolle durch Aufteilung des Arbeits- bzw. Projektverlaufs in überprüfbare Etappen mit Zwischenzielen
  - Umsetzung der Arbeitspakete in konkrete Handlungen und Messen anhand von Prüfkriterien
  - ggf. Ableiten einer Prognose für den weiteren Fortschritt bzw. den Endtermin

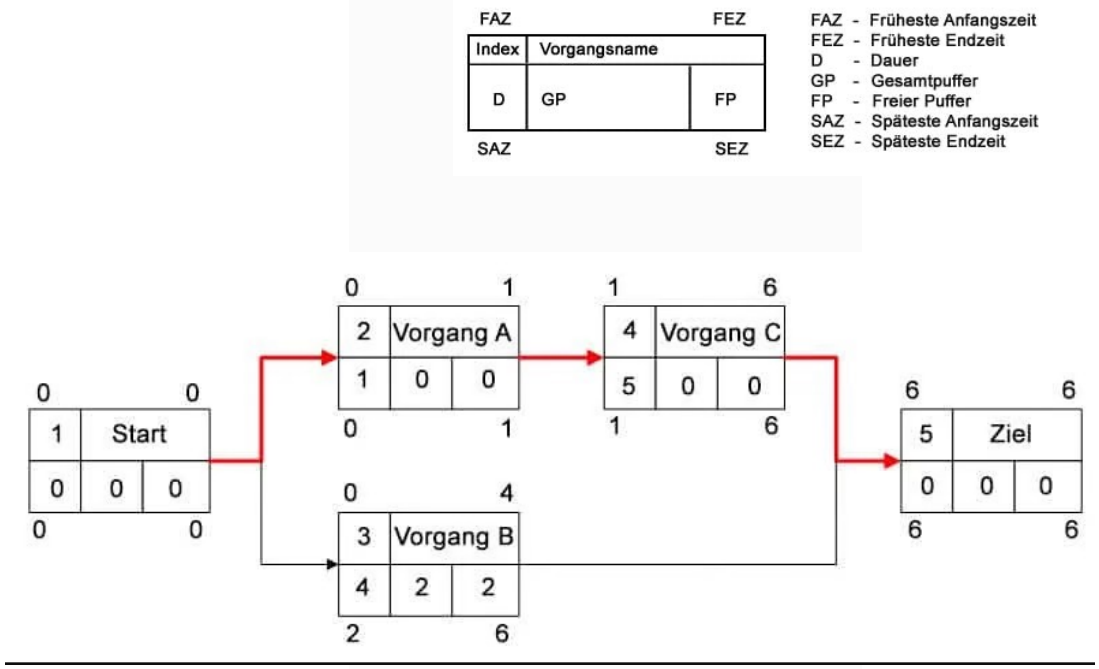
Netzplan

stellt alle Aktivitäten in einer logischen und zeitlichen Abhängigkeit dar

Grundlage für Ressourcenplanung

Darstellung des kritischen Pfades

Gesamtdauer und Pufferzeiten ablesbar



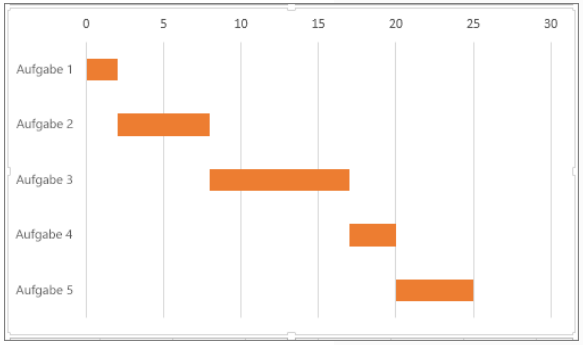
Quelle

Gantt Chart

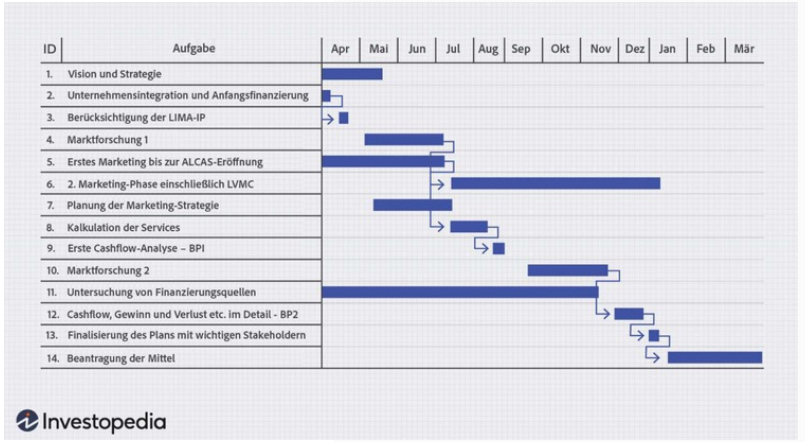
Zeitbezogene Darstellung der Aktivitäten (Aufgaben und Ereignisse)

Meilensteine

Start- und Endtermine der Aufgaben ersichtlich



Quelle



Quelle

Bedarfsanalyse

- Methode, mit der die spezifischen Kundenbedürfnisse ermittelt werden

- **Ziel:**
  - Zielgruppe Angebot unterbreiten, das den Bedarf stillt
- **Methoden**
  - Gezielte Fragen
  - Aktives Zuhören
  - Beispiele: Fragebogen/Interview
- **Inhalt:**
  - Ziel der Kunden
  - IST-Zustand

Analyseverfahren

Verfahren	Beschreibung	Typische Werkzeuge/Notationen
Anforderungsanalyse	Systematische Erfassung aller funktionalen und nicht-funktionalen Anforderungen	Lasten-/Pflichtenheft, User Storys, Use Cases
Ist-Analyse	Untersuchung des aktuellen Systems oder Prozesses, um Verbesserungspotenziale zu erkennen	Prozessdiagramme, Interviews, Beobachtungen
Use-Case-Analyse	Modellierung der Interaktionen zwischen Akteuren (Nutzer/Systeme) und dem System	UML Use-Case-Diagramm
Datenflussanalyse (Data Flow Analysis)	Darstellung, wie Daten im System verarbeitet und transportiert werden	DFD (Data Flow Diagram)
Objektorientierte Analyse (OOA)	Identifikation von Objekten, Klassen und deren Beziehungen im Problemraum	UML-Klassendiagramme
Entity-Relationship-Modellierung (ERM)	Modellierung der Daten und ihrer Beziehungen in einer Datenbank	ER-Diagramme
SWOT-/Risikoanalyse	Bewertung von Stärken, Schwächen, Chancen, Risiken im Projekt	SWOT-Matrix
Gap-Analyse	Vergleich von Ist-Zustand und Soll-Zustand, um Lücken zu identifizieren	Tabellen, Diagramme

Designverfahren

Verfahren	Beschreibung	Typische Werkzeuge/Notationen
Strukturiertes Design (Top-Down)	Zerlegung des Systems in hierarchische Module von oben nach unten	Strukturdiagramme, Pseudocode
Objektorientiertes Design (OOD)	Umsetzung der OOA-Ergebnisse in Klassenstrukturen, Methoden, Schnittstellen	UML-Klassen-, Sequenz- und Aktivitätsdiagramme
Schichten-/Layer-Design	Aufteilung in logische Schichten (z. B. Präsentation, Logik, Datenhaltung)	Architekturdiagramme
Komponentenbasiertes Design	Entwicklung wiederverwendbarer, lose gekoppelter Software-Komponenten	UML-Komponentendiagramme
Serviceorientiertes Design (SOAD)	Entwurf von Services und Schnittstellen nach SOA-Prinzipien	API-Definitionen
Datenbankdesign	Physisches und logisches Design von Datenbanken aus dem ERM	ER-Diagramme, Normalisierungstabellen
UI/UX-Design	Gestaltung der Benutzeroberfläche und Nutzerinteraktion	Wireframes, Mockups
Prototyping	Schnelle Erstellung von Vorabversionen zum Testen von Ideen	Prototypen
Design Patterns	Einsatz bewährter Lösungsbausteine für wiederkehrende Probleme	Singleton, MVC, Observer etc.

Lasten- und Pflichtenheft

	Lastenheft	Pflichtenheft
Urheber	Erstellt durch Auftraggebende (Kundschaft)	Erstellt durch Auftragnehmende
Sichtweise	Kundensicht	Systemsicht/Techn. Sicht
Zweck	Basis für Realisierung eines Projektes seitens Auftragnehmende	• Basis für Auswahl der Angebote seitens Auftraggebende
Inhalt	• Gewünschten Funktionen • Gewünschte Eigenschaften • Erwartete Leistungen • Benötigte Schnittstellen • Geforderte Qualitätskriterien	• Soll-Zustände auf Prozessebene • Tätigkeiten und Mittel der Umsetzung • Technischen Möglichkeiten • Funktionen und Konfigurationen • Ziele/Verpflichtungen
Typische Infos	• Kontextinformationen (z. B. Unternehmensgröße, Standorte oder aktuelle IT-Infrastruktur) • Ist-Zustand/Soll-Zustand • Abgrenzung und Schnittstellen zu anderen Projekten/Produkten • Funktionale/Nicht-Funktionale Anforderungen • Rahmenbedingungen • Vorstellungen bzgl. der Terminierung/Meilensteine	• Kurzüberblick/Ziele • Anforderungen aus Lastenheft • Lösungsbeschreibung • Schnittstellenbeschreibungen • Zeitplan inkl. Meilensteine • Lieferumfang/Abnahmekriterien • Rahmenbedingungen

Funktionale und nicht-funktionale Anforderungen

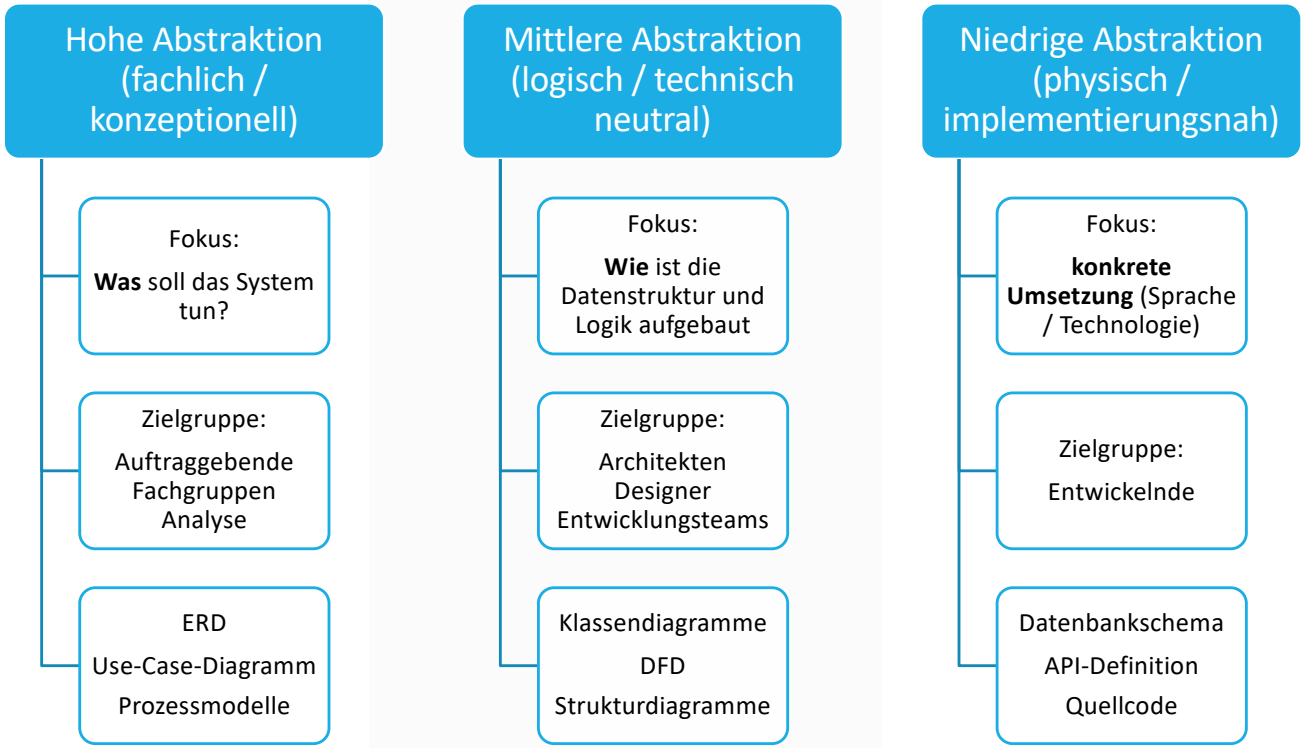
Funktionale Anforderungen

- Anforderungen an eine Anwendung, die beschreiben, was die Anwendung tun soll
- Funktionen einer Anwendung
- Benutzerinteraktionen

Nicht-funktionale Anforderungen

- Betreffen nicht direkt die Funktionalität der Anwendung
- Z. B.: Performance, Sicherheit, Skalierbarkeit

Abstraktionsniveau von Spezifikationen



Spezifikationsinhalte

DATENSTRUKTUREN

- Datentypen (einfach & komplex)
- Beziehungen (1:1, 1:n, m:n)
- Schlüssel (Primary Key, Foreign Key)
- Einschränkungen (Constraints, Validierungen)
- Semantik der Daten

PROGRAMMSTRUKTUREN

- Module/Komponenten
- Schnittstellen und Abhängigkeiten
- Eingaben/Ausgaben
- Kontrollfluss (Ablauflogik)
- Fehlerbehandlung

Angebotsbewertung

- Kosten-Nutzen-Vergleich (Wirtschaftlichkeit)
- Zeitlicher Aufwand
- Technische Anforderungen
- Rechtliche Rahmenbedingungen
- Organisatorische Anforderungen
- Qualität
- Bertungen
- Marktanteil

Qualitativer und quantitativer Angebotsvergleich

Quantitativ	Qualitativ
• Finanzielle Faktoren • einzelnen Faktoren nach einem bestimmten Schema gewichtet und in eine Gesamtbewertung überführt • Objektive Bewertung möglich • Komplex	• nicht-finanzielle Faktoren • Lieferbedingungen • Zuverlässigkeit • Produktqualität und Nachhaltigkeit • Kundenservice • Qualität

Eigenfertigung vs. Fremdbezug

- Entscheidung, ob eine Leistung bzw. ein Produkt intern produziert oder von einem externen Anbietenden bezogen wird
- Berücksichtigung verschiedene Faktoren (Kosten, verfügbaren Ressourcen, Kompetenz des Unternehmens etc.)

Vorteile der Eigenfertigung:
• Mehr Kontrolle über den Produktionsprozess und die Qualität • Größere Flexibilität bei Änderungen oder Anpassungen der Produktion • Schutz von Unternehmens-Know-how und geistigem Eigentum

Vorteile des Fremdbezugs:
• Geringere Investitions- und Fixkosten • Höhere Spezialisierung und Kompetenz des Anbieters • Fokus auf Kernkompetenzen und Auslagerung von nicht-kerngeschäftlichen Aufgaben

Übersicht

Informations- und Hinweispflichten

- Informations- und Hinweispflichten
- Produktbezogene Informationspflichten
- Namens- und Markenrechte
- Urheber- und Nutzungsrechte
- Persönlichkeitsrechte
- Wettbewerbsrecht/Unlauterer Wettbewerb

Informations- und Hinweispflichten

- **Produkte rechtssicher vertreiben und Nutzende korrekt informieren**

Produktbezogene Informationspflichten

Namens- und Markenrecht

Urheber- und Nutzungsrecht

Persönlichkeitsrecht

Wettbewerbsrecht / Unlauterer Wettbewerb

Produktbezogene Informationspflichten

- **Ziel:** Transparenz für Nutzende und Kundschaft
- **Umfang:**

Produktbeschreibung
- Funktionalität, Systemanforderungen, Version

Lizenzinformationen
- z. B. GPL, MIT, proprietär

Bekannte Einschränkungen oder Risiken
- z. B. Beta-Status, Sicherheitslücken

Support- und Update-Informationen

Preisangaben & Vertragsbedingungen
- inkl. Widerrufsrecht bei Verbrauchern nach BGB

Namens- und Markenrecht

- **Rechtsgrundlage:** Markengesetz (MarkenG), BGB

- **Pflichten:**

Vor der Veröffentlichung prüfen, ob Produkt- oder Firmenname markenrechtlich geschützt ist	Keine Verwendung verwechselbarer Namen oder Logos	Eigene Marken ggf. anmelden (DPMA, EUIPO, WIPO)	Hinweis auf eingetragene Marken (® oder ™)

Urheber- und Nutzungsrecht

- **Rechtsgrundlage:** Urheberrechtsgesetz (UrhG)

- **Pflichten:**

Quellcode, Dokumentationen, Grafiken, Sounds sind urheberrechtlich geschützt – auch intern	Nutzung fremder Inhalte (Bibliotheken, Frameworks, Bilder) nur mit gültiger Lizenz	Lizenztexte und -hinweise beilegen (z. B. bei Open Source)	Nutzungsrechte eindeutig im Vertrag regeln (einfach, exklusiv, zeitlich/örtlich beschränkt)

Persönlichkeitsrecht

- **Rechtsgrundlage:** Art. 1 & 2 GG, KunstUrhG, DSGVO

- **Pflichten:**

Keine Veröffentlichung personenbezogener Daten ohne Einwilligung

Schutz der Privatsphäre durch Privacy by Design und Privacy by Default

Bei Verwendung von Bildern oder Videos von Personen: Zustimmung einholen

(Recht am eigenen Bild)

Klare Datenschutzerklärung bereitstellen

(Art. 13 DSGVO)

Wettbewerbsrecht/Unlauterer Wettbewerb

- **Rechtsgrundlage:** Art. 1 & 2 GG, KunstUrhG, DSGVO

- **Pflichten:**

Keine irreführende Werbung (z. B. falsche Leistungsangaben)	Keine Herabsetzung von Konkurrenzprodukten	Keine unzulässige Nachahmung von Konkurrenz-UI/Design (falls schutzfähig)	Pflicht zur klaren Preisangabe bei Verkauf (inkl. Steuern, Zusatzkosten)

Übersicht

Grundlagen von Programmiersprachen

- Identifikation und Auswahl einer passenden Programmiersprache
- Programmierparadigmen
- Imperativ versus deklarativ
- Skriptsprachen
- GPL versus DSL
- Abstraktionsniveau von Sprachen
- Compiler, Linker, Interpreter
- Typisierung
- Variablen, Datentypen und -strukturen, Zuweisungen
- Primitive Datentypen
- Programmstrukturen
- Kontrollstrukturen
- Prozeduren versus Funktionen
- Rekursion
- Speicherbedarf und Performance bekannter Sprachen im Vergleich
- Portabilität

Identifikation und Auswahl einer passenden Programmiersprache

Problem:

- Welche Anforderungen hat das Problem? Welche Funktionalität muss die Lösung bereitstellen?

Team:

- Welche Fähigkeiten und Erfahrungen hat das Team in verschiedenen Programmiersprachen?

Ressourcen:

- Welche Ressourcen sind verfügbar, um das Problem zu lösen (z.B. Budget, Zeit)?

Skalierbarkeit:

- Wird die Lösung auf lange Sicht wachsen oder sich ändern müssen? Welche Skalierbarkeit bietet die ausgewählte Sprache?

Performance:

- Wie wichtig ist die Performance der Lösung? Welche Sprache bietet die beste Performance für das Problem?

Verfügbarkeit von Tools und Bibliotheken:

- Welche Tools und Bibliotheken gibt es für die ausgewählte Sprache, die bei der Lösung des Problems helfen könnten?

Paradigma	Eigenschaften	Beispiel
Imperative Programmiersprachen	Folge von Befehlen, die vorgeben, in welcher Reihenfolge was vom Computer getan werden soll **WIE** soll etwas berechnet werden	
➢ Strukturierte Programmiersprachen	Verwendete Kontrollstrukturen: Anweisungsreihenfolge, Verzweigung und Schleifen	Ada, C
➢ Prozedurale Programmiersprachen	Zerlegung der Programme in kleinere Teilaufgaben (Prozeduren)	Fortran, COBOL, Pascal
Deklarative Programmiersprachen	**WAS** soll berechnet werden Nicht der Lösungsweg wird programmiert, sondern welches Ergebnis gewünscht ist	
➢ Funktionale Programmiersprachen	Funktionen werden definiert, angewendet und miteinander wie miteinander verknüpft, Sie können als Parameter verwendet werden und als Funktionsergebnisse auftreten	Haskell, F#
➢ Logische Programmiersprachen	Beschreibung von Fakten als wahre Aussagen und Regeln, meist Prädikatenlogik erster Stufe Variablen werden automatisch mit passenden Werten gebunden, die eine Lösung ermöglichen Keine explizite Steuerung von Kontrollfluss oder Speicherverwaltung nötig	Prolog, Datalog, Mercury, Curry
Objektorientierte Programmiersprachen	Klassen als Grundelemente Objekte werden mit Daten und den darauf arbeitenden Routinen zu Klassen zusammengefasst Datenkapselung, Vererbung, Polymorphie	C#, Java

Programmierparadigmen

Imperativ versus deklarativ

Merkmal	Imperativ	Deklarativ
Fokus	**Wie** etwas getan wird	**Was** erreicht werden soll
Steuerung	Programmierer gibt Ablauf vor	Interpreter/Compiler entscheidet Ablauf
Zustand	Häufig veränderlich	Meist zustandslos oder unveränderlich
Lesbarkeit	Detailreich, oft länger	Kürzer, ausdrucksstark
Beispiele	C, Java (imperativ genutzt), Python (klassisch)	SQL, HTML, Haskell, funktionale JS-Methoden
Typische Nutzung	Algorithmen mit komplexer Ablaufsteuerung	Datenabfragen, Beziehungen, Transformationen, UI-Beschreibung

- Viele Sprachen wie Python, JavaScript, Java, C# unterstützen beide Ansätze
- Funktionale Features (z. B. map, filter, reduce, Lambdas) fördern den deklarativen Stil
- Kontrollstrukturen und klassische Methodenaufrufe folgen dem imperativen Stil

Skriptsprachen

- Programmiersprachen, die mit einem Interpreter „on-the-Fly“ ausgeführt werden
- Kompilieren vor Ausführung nicht notwendig
- Werden „interpretiert“

- **Beispiele:**
  - Shell-Skripte - Programme, die in der Unix-Shell ausgeführt werden

- **Anwendung:**
  - wiederkehrende Systemabläufe automatisieren und überwachen
  - Optimieren und Automatisieren lokaler und netzwerkübergreifender Aufgaben

GPL versus DSL

GENERAL PURPOSE LANGUAGE

- Programmiersprache für eine Vielzahl unterschiedlicher Anwendungen
- Bietet allgemeine Kontrollstrukturen, Datentypen und Bibliotheken für viele Szenarien
- **Eigenschaften**
  - Vielseitig einsetzbar
  - umfangreiche Tool-Unterstützung
  - Oft mit breiten Standardbibliotheken
- **Beispiele**
  - Python, Java, C, C++, JavaScript, Rust

DOMAIN SPECIFIC LANGUAGE

- Programmiersprache speziell für einen bestimmten Anwendungsbereich (Domäne)
- Enthält nur relevante Sprachkonstrukte
- **Eigenschaften**
  - Optimiert für bestimmte Problemstellungen
  - Oft einfacher zu lesen und schreiben als GPLs
  - Interne DSL – eingebettet in eine GPL (z. B. SQL-Queries in Python)
  - Externe DSL – eigene Syntax, eigener Parser (z. B. CSS, HTML)
- **Beispiele**
  - SQL, HTML/CSS, VHDL, Reguläre Ausdrücke (Regex), MATLAB

Abstraktionsniveau von Sprachen

1GL – First Generation Language (*Maschinensprache*)

direkt ausführbarer **Binärcode**, bestehend aus 0 und 1

Eigenschaften:

- Extrem hardwarenah
- Prozessorabhängig (Befehlssatz)
- Schwer lesbar und fehleranfällig
- Höchste Ausführungs-geschwindigkeit, keine Übersetzung nötig

2GL – Second Generation Language (*Assembler/Assemblersprache*)

Symbolische Darstellung der Maschinensprache

Eigenschaften:

- Immer noch hardwarenah
- Ein Mnemonik-Befehl entspricht meist genau einem Maschinenbefehl
- Prozessorabhängig
- Assembler übersetzt Code in Maschinensprache

3GL – Third Generation Language (*Höhere Programmiersprachen*)

Höher abstrahiert

Eigenschaften:

- Maschinunabhängig
- imperativ oder objektorientiert
- Lesbarer, an natürlicher Sprache angelehnt
- Compiler oder Interpreter übersetzt in Maschinensprache
- Portabel zwischen verschiedenen Systemen
- Unterstützt Strukturen wie Schleifen, Funktionen, Klassen.

4GL – Fourth Generation Language (*Problemorientierte Sprachen*)

Sehr hoch abstrahiert, oft domänenspezifisch, mit Fokus darauf, *was* erreicht werden soll, nicht *wie*

Eigenschaften:

- Wenig Code, oft deklarativ
- Starke Automatisierung durch Tools
- Häufig in Datenbanken, Reporting, Statistik oder Konfigurationssystemen

Compiler, Linker, Interpreter

Compiler

- Programme, die Quellcode in ausführbaren Maschinencode übersetzen
- **Nachteil**: lange Compilierzeiten bei großen Projekten
- **Vorteil**: hohe Ausführungsgeschwindigkeit

Linker

- Programme, die mehrere, in Maschinencode übersetzte Programme miteinander verbinden
- Werden verwendet, um Programme aus mehreren Dateien zusammenzufügen und sicherzustellen, dass alle benötigten Ressourcen verfügbar sind

Interpreter

- Programme, die Quellcode „on-the-fly“ (Zeile für Zeile) in Maschinencode übersetzen und direkt ausführen
- **Nachteil**: langsamer als Compiler
- **Vorteil**: Fehler werden zeilenweise angezeigt

Typisierung 1/2

STATISCH

- Datentyp einer Variable zur Compile-Zeit festgelegt und geprüft
- Typfehler werden früh erkannt
- Variablen müssen oft mit Typangabe deklariert werden
- Mehr Sicherheit, weniger Flexibilität

DYNAMISCH

- Datentyp einer Variable zur Laufzeit bestimmt
- Keine (oder wenige) Typdeklarationen nötig
- Mehr Flexibilität
- Typfehler treten erst bei Ausführung auf

Typisierung 1/2

STARK

- Datentypen werden strikt eingehalten
- automatische Umwandlungen (Implizite Casts) kaum oder gar nicht möglich
- Mehr Sicherheit, weniger ungewollte Typkonvertierungen
- Erfordert oft explizites Casten

SCHWACH

- Das System konvertiert Datentypen automatisch (oft stillschweigend), um Operationen durchzuführen
- Mehr Flexibilität
- Höheres Risiko für unerwartete Ergebnisse

Variablen, Datentypen und -strukturen, Zuweisungen

Variablen

- Behälter, die für Werte verwendet werden
- besitzen einen Namen und einen bestimmten Datentyp

Datentypen

- Beschreiben, welche Art von Wert eine Variable haben kann

Datenstrukturen

- Spezielle Arten von Datentypen, die es ermöglichen, komplexere Daten zu organisieren und zu verarbeiten
- Beispiel: Listen, Arrays

Zuweisungen

- Anweisungen, die verwendet werden, um einer Variablen einen Wert zuzuweisen

Primitive Datentypen

- byte:
  - Speichert 8-Bit-Ganzzahl
  - Wertebereich: -128 bis 127
- short:
  - Speichert 16-Bit-Ganzzahl
  - Wertebereich: - 32.768 bis 32.767
- int:
  - Speichert 32-Bit-Ganzzahl
  - Wertebereich: - 2.147.483.648 bis 2.147.483.647
- long:
  - Speichert 64-Bit-Ganzzahl
  - Wertebereich: - 2^63 bis 2^63-1
- float:
  - Speichert 32-Bit-Fließkommazahl
  - Wertebereich und Genauigkeit sind abhängig von der Implementierung
- double:
  - Speichert 64-Bit-Fließkommazahl
  - Wertebereich und Genauigkeit sind abhängig von der Implementierung
- boolean:
  - Speichert entweder true oder false
- char:
  - Speichert einzelnes Zeichen

Programmstrukturen

Lineare Strukturen:

- Folge von Anweisungen, die in der angegebenen Reihenfolge ausgeführt werden

Hierarchische Strukturen:

- Programme werden in Module, Funktionen oder Klassen unterteilt, die hierarchisch angeordnet sind

Modulare Strukturen:

- Programme werden in wieder-verwendbare Module oder Komponenten zerlegt

Kontroll-strukturen

Sequenz

- Folge von Anweisungen

Verzweigungen

- Beeinflussen den Ablauf eines Programms an bestimmten Stellen
- Häufigste Verzweigungen:
  - **if-Anweisungen**
  - **switch-Anweisungen**

Wiederholungen (auch als „Schleifen“ bezeichnet)

- Ermöglichen Mehrfache Ausführung bestimmter Programmteile
- Arten
  - Kopfgesteuerte Wiederholungen:
    - Zu Beginn einer Schleife festgelegt, wie oft die Schleife ausgeführt wird
    - **while-Schleifen, for-Schleifen**
  - Fußgesteuerte Wiederholungen:
    - Am Ende einer Schleife festgelegt, ob die Schleife wiederholt wird oder nicht
    - **until-Schleifen**

Prozeduren versus Funktionen

Prozeduren	Funktionen
• Abschnitte eines Programms, die aus einer Reihe von Anweisungen bestehen • Von anderen Teilen des Programms aufrufbar	• Ähnlich den Prozeduren • Geben einen Wert zurück

Rekursionen 1/2

- Funktion ruft sich selbst (direkt oder indirekt) auf, um ein Problem zu lösen
- Das große Problem wird dabei in kleinere Teilprobleme zerlegt, bis eine Abbruchbedingung (Basisfall) erreicht wird

- **Basisfall** (Abbruchbedingung):
  - Stoppt die Rekursion, verhindert Endlosschleifen
- **Rekursiver Fall**:
  - Funktion ruft sich mit einem "kleineren" Problem auf
- **Stack-Verwaltung**:
  - Jeder Funktionsaufruf wird auf den **Call Stack** gelegt
  - Nach Erreichen des Basisfalls werden die Aufrufe wieder abgearbeitet (**Unwinding**)

```python
def fakultaet(n):
    if n == 0 or n == 1:  # Basisfall
        return 1
    else:
        return n * fakultaet(n - 1)  # Rekursiver Fall

print(fakultaet(5))  # 120
```

Rekursion 2/2

VORTEILE

- **Eleganz & Verständlichkeit**:
  - Viele Algorithmen (z. B. Divide-and-Conquer-Algorithmen) lassen sich sehr einfach rekursiv beschreiben
- **Kürzerer Code**:
  - Weniger Schleifen
  - übersichtlichere Implementierung
- **Natürliche Abbildung**:
  - hierarchischer oder selbstähnliche Probleme (Bäume, Graphen, Fibonacci)

NACHTEILE

- **Performanceprobleme**:
  - Jeder Aufruf braucht Stack-Speicher → Gefahr von **Stack Overflow bei** zu vielen Rekursionstiefen
- **Overhead**:
  - Jeder Funktionsaufruf kostet mehr als eine Schleifeniteration
- **Komplexität**:
  - Für manche Probleme ist iterative Lösung effizienter
- **Fehlersuche**:
  - Tiefe Rekursionen sind schwieriger zu verfolgen

Speicherbedarf im Vergleich

Merkmal	C	Java	JavaScript
Speicher-verbrauch	Gering – sehr effizient, da man selbst kontrolliert	Höher – JVM benötigt zusätzlichen Overhead	Hoch – Browser/Node.js-Laufzeitumgebung benötigt viel Speicher
Speicher-verwaltung	Manuell – volle Kontrolle, aber fehleranfällig (Memory Leaks, Buffer Overflows)	Automatisch – Garbage Collector kümmert sich um Speicherfreigabe	Automatisch – Garbage Collector kümmert sich um Speicherfreigabe
Eignung für Embedded/Low-Memory-Geräte	Sehr gut geeignet	Weniger geeignet (großer VM-Overhead)	Kaum geeignet (große Laufzeitumgebungen)

Performance im Vergleich

Merkmal	C	Java	JavaScript
Performance	Sehr hoch – wird direkt in Maschinencode kompiliert	Hoch – Bytecode wird in JVM JIT kompiliert	Mittel – wird interpretiert, moderne Engins nutzen JIT-Optimierung
Grund	Native Ausführung ohne Laufzeit-VM	Just-In-Time-Compiler der JVM optimiert Code zur Laufzeit	JIT-Compiler in Browsern (z. B. V8) beschleunigen stark, aber nicht auf C-Niveau
Einsatz bei Performance-kritischen Anwendungen	Sehr gut geeignet (z. B. Spiele-Engines, Treiber, Compiler)	Gut geeignet, wenn Plattformunabhängigkeit gewünscht ist	Nur eingeschränkt geeignet, eher für Interaktivität und Web-Frontends

Portabilität

- **Java**: Sehr hohe Portabilität, Code in Bytecode kompiliert, läuft auf jeder Plattform mit JVM
- **C**: Portabel, wenn nur Standardbibliotheken genutzt werden – plattformspezifische Anpassungen können nötig sein
- **Python**: Hoch portabel, wenn keine OS-spezifischen Module genutzt werden

- Ausführung eines Programms auf unterschiedlichen Betriebssystemen und/oder Hardwarearchitekturen

- **Faktoren**
  - Standardisierung der Programmiersprache
  - Verwendung plattformunabhängiger Bibliotheken
  - Abstraktion von Betriebssystemfunktionen
  - Vermeidung proprietärer Funktionen
  - Cross-Platform-Tools

- **Vorteile**
  - Geringerer Entwicklungsaufwand
  - Größere Zielgruppe
  - Bessere Zukunftssicherheit, falls Plattformen wechseln

Übersicht

Funktionale Programmierung

- Eigenschaften Funktionaler Programmierung
  - Functions as First Class Citizens
  - Pure Functions
  - Higher Order Functions (HOFs)
  - Immutability
  - Fokus auf Rekursion (TCO)
  - Pattern Matching

Eigenschaften funktionaler Programmierung

- Functions as „First Class Citizens“
- Pure Functions
- Higher Order Functions
- Immutability
- Fokus auf Rekursion (Tail Call Optimization)
- Pattern Matching

Functions as First Class Citizens

- Funktionen werden wie jede andere Variable behandelt:
  - in Variablen speichern
  - als Argument übergeben
  - als Rückgabewert

JavaScript

```
function add(a, b) { return a + b; }
const op = add;           // Funktion in Variable speichern
function apply(fn, x, y){ return fn(x, y); }
console.log(apply(op, 2, 3)); // 5
```

Pure Functions

- Gleiche Eingabe → immer gleiche Ausgabe
- Keine Nebeneffekte (z. B. keine Änderung globaler Variablen)
- **Vorteile**:
  - Einfach zu testen
  - Vorhersagbares Verhalten

Python

```
def pure_add(a, b):
    return a + b        # gleiche Eingabe -> gleiche Ausgabe, keine Seiteneffekte
print(pure_add(3, 4))  # 7
```

Higher Order Functions (HOFs)

- Funktionen, die andere Funktionen als Parameter nehmen oder zurückgeben

JavaScript

```
const multiplyBy = factor => x => x * factor;
const double = multiplyBy(2);
console.log(double(5)); // 10
```

Immutability

- Einmal erstellte Datenstrukturen werden nicht verändert
- bei Änderungen wird neue Version erstellt
- **Vorteil**:
  - Verhindert unerwartete Nebeneffekte
  - erleichtert parallele Ausführung

JavaScript

```
const arr = [1,2,3];
const newArr = [...arr, 4];   // erzeugt neue Struktur
// arr bleibt [1,2,3], newArr ist [1,2,3,4]
```

Fokus auf Rekursion (Tail Call Optimization)

- Statt Schleifen wird Rekursion genutzt
- Tail Call Optimization (TCO) optimiert rekursive Funktion so, dass kein zusätzlicher Stackframe erzeugt wird (Vermeidung von Stack Overflow)

Python

```
def factorial(n, acc=1):
    if n == 0:
        return acc
    return factorial(n-1, acc*n)

print(factorial(5))
```

Pattern Matching

- Strukturierte Fallunterscheidung, ähnlich einem erweiterten switch
- oft mit Destrukturierung

```
Python 3.10+
def describe(x):
    match x:
        case 0:           return "Zero"
        case 1:           return "One"
        case int():       return "Some Integer"
        case _:           return "Something else"
print(describe(5))  # Some Integer
```

Übersicht

Objektorientierung

- Klassen – Eigenschaften
- Klassenbeziehungen
- Polymorphie
- Generische Klassen
- Statische versus nicht-statische Methoden
- Datenstrukturen
- Funktionale Ausdrücke in modernen Sprachen

Klassen - Eigenschaften

Klasse

- „Vorlage“, die Eigenschaften und Verhaltensweisen eines Objekts festlegt

Objekte

- Instanzen von Klassen

Methoden

- Funktionen/Prozeduren eines Objektes

Attribute

- Eigenschaften eines Objektes

Sichtbarkeit

- public – Element ist von anderen Klassen aus sichtbar und aufrufbar
- protected – Element ist nur innerhalb derselben Klasse und der erweiternden Klassen sichtbar und aufrufbar
- private – Element ist nur innerhalb des definierenden Bereichs sichtbar und aufrufbar

Klassenbeziehungen

- Assoziation
- Aggregation
- Komposition
- Spezialisierung
- Generalisierung

Assoziation

- Beziehung zwischen zwei oder mehr Klassen

- Beispiel:
  - Eine Person hat ein oder mehrere Konto/Konten

Aggregation

- Beziehung zwischen einem Ganzen und seinen Teilen
  - „Besteht-aus“-Beziehung
- Klasse kann unabhängig von einer anderen Klasse existieren

- Beispiel:
  - Eine Vorlesung „besteht aus“ Studierende
  - Studierende können auch ohne Vorlesungen existieren

Komposition

- Spezialfall der Aggregation
- Teile hängen von der Existenz des Ganzen ab

- Beispiel:
  - Ein Gebäude „besteht aus“ Räumen
  - Ohne dem Gebäude kann der Raum nicht existieren

Generalisierung

- Gerichtete Beziehung zwischen einer generelleren und einer spezielleren Klasse
- Exemplare der spezielleren Klasse sind Exemplare der generelleren Klasse
- Speziellere Klasse verfügt implizit über alle Merkmale (Struktur- und Verhaltensmerkmale) der generelleren Klasse

Interfaces

- Schablone für eine Klasse
- definieren Methoden-Signaturen, die von nicht-abstrakten Klassen konkret realisiert werden müssen
- Nicht instanziierbar
- Keine Attribute

Polymorphie (Vielgestaltigkeit)

- Ableitung mehrerer Klassen von einer gemeinsamen Basisklasse (z. B. Interface)
- Objekt kann mehrere Formen annehmen
- abgeleitete Klassen implementieren ein und dieselbe Methode
- bei Aufruf dieser Methode wird konkrete Implementierung der entsprechenden Klasse verwendet

- **Zweck:**
  - einheitliche Schnittstellen verwenden
  - unterschiedliche Implementierungen bereitstellen

Vererbung

- Klasse (Subklasse, Kindklasse) erbt Eigenschaften und Methoden einer anderen Klasse (Superklasse, Elternklasse)
- **Zweck:**
  - Wiederverwendung von Code
  - Hierarchien abbilden
  - Gemeinsames Verhalten zentral definieren
- **Merkmale:**
  - Kindklasse kann Methoden und Attribute der Elternklasse **übernehmen**
  - Kindklasse kann geerbte Methoden **überschreiben** (*Overriding*) und **überladen** (*Overloading*)

Vererbung versus Polymorphie

Merkmal	Vererbung	Polymorphie
Definition	Beziehung zwischen Klassen (Eltern-Kind)	Unterschiedliche Implementierungen hinter derselben Schnittstelle
Fokus	Codewiederverwendung, Hierarchie	Austauschbarkeit von Objekten zur Laufzeit
Bindung	Struktur zur Compile-Zeit	Verhalten wird oft erst zur Laufzeit entschieden

Generische Klassen

- Klassen, die Typparameter verwenden, um unabhängig vom konkreten Datentyp zu arbeiten
- gleicher Code kann mit unterschiedlichen Datentypen genutzt werden, ohne dass er mehrfach für jeden Typ neu geschrieben werden muss
- Funktionsweise
  - Klasse definiert Typparameter `<T>`, welcher später bei der Instanziierung der Klasse durch einen konkreten Typ ersetzt wird
  - `T` ist Platzhalterbezeichnung

Vorteile

- Wiederverwendbarkeit
- Typsicherheit: Fehler werden schon zur Compile-Zeit erkannt, da der verwendete Typ bekannt ist
- Wegfall von Casts: Umwandlung zwischen Typen entfällt

Vorteile generischer Klassen ggü. Arrays

Kriterium	Generische Container (Templates)	Arrays
Dynamische Größe	Größe kann zur Laufzeit wachsen oder schrumpfen	Feste Größe nach Deklaration; Änderung erfordert neue Speicherallokation
Typsicherheit	Typ wird durch Template-Parameter festgelegt → keine impliziten Typkonvertierungen nötig	Kann Typfehler verursachen, besonders bei C-Arrays mit void-Pointern oder Casts
Zusatzfunktionen	Viele Methoden für Einfügen, Löschen, Suchen, Sortieren, Iterieren	Keine eingebauten Methoden – alles muss manuell implementiert werden
Sichere Speicherverwaltung	Speicherverwaltung ist gekapselt; Gefahr von Speicherlecks geringer	Entwickelnde müssen Speicherverwaltung selbst übernehmen
Iterator-Unterstützung	Einheitliche Schnittstellen für Iteration	Iteration nur per Index
Flexibilität der Datenstruktur	Verschiedene Containerarten verfügbar: Vektor, Liste, Set, Map – Auswahl je nach Anwendungsfall	Immer nur flacher, zusammenhängender Speicherblock
Einfachere Erweiterbarkeit	Lesbarer, wiederverwendbarer Code; weniger Fehlerquellen	Mehr Boilerplate-Code; Fehleranfälliger

Statische versus nicht-statische Methoden

STATISCH

- Gehören zur Klasse selbst
- **Aufruf:** ohne Instanz der Klasse
- **Zugriffsmöglichkeiten:**
  - Zugriff nur auf statische Attribute und Methoden
  - kein Zugriff auf Instanzvariablen
- **Verwendung:**
  - Konstanten und Zähler für Instanzen
  - Fabrikmethoden (Factory Pattern)
  - Mathematische oder Utility-Funktionen

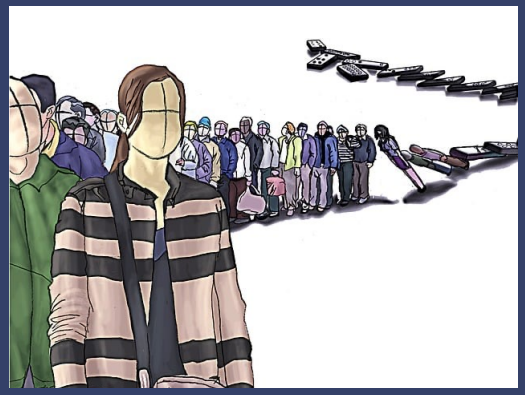
NICHT-STATISCH (DYNAMISCH)

- Gehören zu einem bestimmten Objekt (Instanz)
- **Aufruf:** Benötigen eine Instanz der Klasse
- **Zugriffsmöglichkeiten:**
  - Zugriff auf alle Attribute und Methoden der Instanz
  - Zugriff auf statische und nicht-statische Mitglieder
- **Verwendung:**
  - Immer dann, wenn der Objektzustand (Instanzvariablen) relevant ist

Datenstrukturen

Struktur	Organisation	Typische Zugriffsregel	Typische Verwendung
Queue	Linear	FIFO	Warteschlangen, Nachrichten
Stack	Linear	LIFO	Funktionsaufrufe, Undo
Array	Linear	Indexzugriff	Listen, Tabellen
Baum	Hierarchisch	Eltern-Kind-Beziehung	Dateisysteme, Suchstrukturen
Heap	Baumförmig	Max-/Min-Eigenschaft	Priority Queues, Sortieren
Graph	Netzartig	Knoten & Kanten	Netzwerke, Pfadsuche

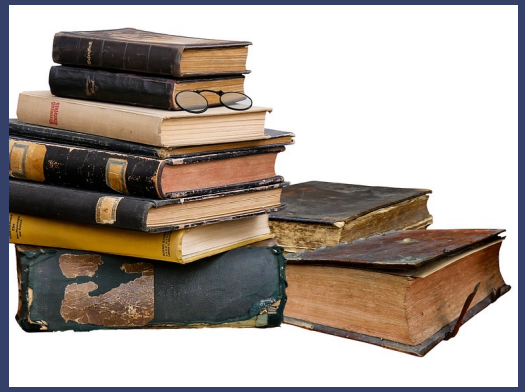
Queue



Prinzip: First In – First Out (FIFO)

- **Operationen:**
  - enqueue (hinzufügen am Ende)
  - dequeue (entfernen vom Anfang)
  - peek (erstes Element ansehen, ohne zu löschen)
- **Einsatzbeispiele:**
  - Druckerwarteschlangen
  - Aufgabenverwaltung
  - Nachrichtenverarbeitung
- **Varianten:**
  - Priority Queue (Elemente mit Priorität)
  - Circular Queue (Ringspeicher)

Stack



Prinzip: Last In – First Out (LIFO)

- **Operationen:**
  - push (hinzufügen oben)
  - pop (entfernen oben)
  - peek (oberstes Element ansehen)

- **Einsatzbeispiele:**
  - Funktionsaufruf-Stack
  - Undo/Redo-Funktionalität
  - Klammerauswertung

Array

Prinzip: Feste Anzahl von Elementen, die im Speicher zusammenhängend liegen

- **Zugriff:** Direkter Zugriff über Index (O(1))
- **Vorteile:** Sehr schnelle Indexoperationen
- **Nachteile:**
  - Feste Größe (bei statischen Arrays)
  - teures Einfügen/Löschen in der Mitte
- **Einsatzbeispiele:**
  - Listen von festen Daten
  - Tabellen
  - Matrizen

Tree

Prinzip: Hierarchische Datenstruktur mit Wurzel (Root) und Kindknoten

- **Sonderform: Binärbaum** – jeder Knoten hat maximal 2 Kinder

- **Begriffe:**
  - Elternknoten (Parent),
  - Kindknoten (Child)
  - Blatt (Leaf) – Knoten ohne Kinder
  - Tiefe (Depth) – Abstand zur Wurzel

- **Einsatzbeispiele:**
  - Dateisysteme
  - Entscheidungsbäume
  - Suchbäume

Heap

Prinzip: vollständiger Binärbaum

- **Ordnungsrelation der Elternknoten:**
  - Max-Heap: Eltern ≥ Kinder
  - Min-Heap: Eltern ≤ Kinder

- **Einsatzbeispiele:**
  - Priority Queues
  - Heap-Sort
  - Speicherverwaltung

Graph

Prinzip: Menge von Knoten (Vertices) und Kanten (Edges), die Knoten verbinden

- **Arten:**
  - Gerichtet / Ungerichtet
  - Gewichtet / Ungewichtet
  - Zusammenhängend / Nicht zusammenhängend

- **Einsatzbeispiele:**
  - Soziale Netzwerke
  - Straßennetze
  - Abhängigkeitsdiagramme

- **Darstellung:**
  - Adjazenzmatrix
  - Adjazenzliste

Funktionale Ausdrücke in modernen Sprachen

- Lambda-Ausdrücke
- Functional Interfaces
- Map/Filter/Reduce

Lambda-Ausdrücke

- kurze, namenlose Funktionsdefinitionen
- direkt dort erstellt, wo sie gebraucht werden – ohne vorher eine komplette Methode oder Funktion zu deklarieren
- Für kompakteren, ausdrucksstärkeren Code
- Kann Parameter haben und einen Wert zurückgeben
- Oft in Kombination mit höherwertigen Funktionen (map, filter, reduce)
- Wird wie ein Wert behandelt (z. B. in Variablen gespeichert oder als Argument übergeben)

```python
# Lambda-Ausdruck (anonyme Funktion)
quadrat_lambda = lambda x: x * x

print(quadrat(5))        # 25
print(quadrat_lambda(5)) # 25
```

Python

```java
// Lambda-Ausdruck: (Parameter) -> {Anweisung}
zahlen.forEach(x -> System.out.println(x * x));
```

Java

```csharp
// Lambda als Variable
Func<int, int> quadrat = x => x * x;
Console.WriteLine(quadrat(5)); // 25

// Lambda direkt in LINQ
List<int> zahlen = new List<int> { 1, 2, 3, 4 };
var quadrierteZahlen = zahlen.Select(x => x * x);
```

C#

Functional Interfaces

- Interface mit genau einer abstrakten Methode (C#: Delegates)
- als Ziel für Lambda-Ausdrücke oder Methodenreferenzen
- **Eigenschaften**
  - Hat nur eine abstrakte Methode (Single Abstract Method → SAM)
  - Kann mehrere default- oder static-Methoden enthalten, diese zählen nicht zu den abstrakten Methoden
  - Annotation: @FunctionalInterface in Java (optional, aber empfehlenswert → Compiler prüft Einhaltung)
- **Vorteile**
  - Ermöglicht Einsatz von Lambda-Ausdrücken und Methodenreferenzen
  - Erhöht Lesbarkeit und reduziert Boilerplate-Code
  - Zentrale Grundlage für Streams und funktionale APIs in Java

Functional Interfaces in Java - Beispiel

```java
@FunctionalInterface
interface Rechner {
    int berechne(int a, int b); // einzige abstrakte Methode
}

public class FunctionalInterfaceDemo {
    public static void main(String[] args) {
        // Lambda-Ausdruck als Implementierung
        Rechner addieren = (x, y) -> x + y;
        Rechner multiplizieren = (x, y) -> x * y;

        System.out.println("Addition: " + addieren.berechne(5, 3));     // 8
        System.out.println("Multiplikation: " + multiplizieren.berechne(5, 3)); // 15
    }
}
```

Zentrale Funktionen zur Verarbeitung von Datenkollektionen

MAP

- **Zweck:**
  - Wendet eine Funktion auf jedes Element einer Sammlung an und gibt eine neue Sammlung mit den Ergebnissen zurück
- **Merkmale:**
  - Ändert die Anzahl der Elemente **nicht**
  - Originaldaten bleiben unverändert

FILTER

- **Zweck:**
  - Filtert Elemente basierend auf einer Bedingung (Prädikat) und gibt nur die Elemente zurück, die die Bedingung erfüllen
- **Merkmale:**
  - Die Anzahl der Elemente **kann kleiner** sein
  - Originaldaten bleiben unverändert

REDUCE

- **Zweck:**
  - Führt eine Funktion schrittweise auf die Elemente einer Sammlung an, um sie auf einen einzigen Wert zu reduzieren
- **Merkmale:**
  - Ergebnis ist **kein Array**, sondern z. B. Summe, Produkt oder kombinierter Wert

Übersicht

Asynchrone Programmierung

- Synchrone und asynchrone Programmierung
- Reactive Programming
- Nebenläufigkeit versus Parallelität
- Herausforderungen paralleler Programmierung
  - Deadlocks versus Livelocks
  - Race Conditions versus Starvation
  - Schwierige Fehlerreproduktion, Debugging & Testing
  - Synchronisations-Overhead
  - Cache-Kohärenz und False Sharing

Synchrone und asynchrone Programmierung

Aspekt	Synchron	Asynchron
Definition	Aufgaben laufen nacheinander; jeder Schritt wartet auf den vorherigen	Aufgaben können überlappen; ein Aufruf blockiert nicht den Thread
Kontrollfluss	Linear, leicht nachzuvollziehen	Ereignis-/Callback- oder Future/Promise-getrieben
Thread-Blocking	Häufig blockierend (I/O, Sleep)	Nicht-blockierend (I/O wird ausgelagert, Thread bleibt frei)
Latenz/ Throughput	Höhere Latenz bei I/O; geringer Durchsatz unter Last	Bessere Auslastung bei I/O-lastigen Aufgaben; höherer Durchsatz
Komplexität	Gering; Debugging einfacher	Höher (Callbacks, Promises, async/await, Race Conditions)
Fehlertypen	Klassisch (Exceptions, Deadlocks bei Multi-Threading)	Callback-Hell, Promise-Ketten, Concurrency-Bugs (Race, Starvation)
Einsatz	CPU-lastige Schritte ohne I/O; kleine Skripte, CLIs	Netz-/Datei-I/O, Webserver, GUIs, viele gleichzeitige Clients
Ressourcen	Wenige Abhängigkeiten; kann Threads blockieren → ineffizient	Effizientere Nutzung von Threads/Events/Loops; braucht Laufzeit-Support

Reactive Programming

- Programmierparadigma, das darauf ausgerichtet ist, asynchrone Datenströme und deren Änderungen in Echtzeit zu verarbeiten
- Teile des Programms reagieren automatisch auf Ereignisse oder Datenänderungen

Konzept	Bedeutung
Observable	Quelle von Datenereignissen (z. B. Mausklicks, WebSocket-Daten, API-Responses)
Observer / Subscriber	"Abonnent", der auf Ereignisse aus der Observable reagiert
Operators	Transformationen, Filterungen oder Kombinationen von Datenströmen (map, filter, merge)
Backpressure	Mechanismus, um mit sehr schnellen Datenströmen umzugehen, ohne den Empfänger zu überlasten

Nebenläufigkeit versus Parallelität

Nebenläufigkeit:

- Mehrere Aufgaben teilen sich Ressourcen zeitlich
- kann auch auf einem Kern passieren

Parallelität:

- Aufgaben laufen gleichzeitig auf mehreren Kernen

Konzeptionelle Vorgehensweise beim Parallelisieren von Anwendungen

- 1. Partitionierung
  - Zerlegung einer Gesamtaufgabe in möglichst viele vollständige Teilaufgaben
- 2. Kommunikation
  - Kommunikation-/Datenfluss identifizieren
  - Wo sind Abhängigkeiten (Ergebnis einer Teilaufgabe ist Parameter einer anderen Aufgabe)
  - Welche Teilaufgaben sind unabhängig voneinander
- 3. Zusammenfassung
  - Bündelung von Teilaufgaben
- 4. Verteilung
  - Verteilung von Teilaufgaben auf die zur Verfügung stehenden Kerne

Herausforderungen paralleler Programmierung

- Race Conditions (Wettlaufsituationen)
- Deadlocks (Verklemmungen)
- Livelocks
- Starvation (Verhungern)
- Schwierige Fehlerreproduktion, Debugging & Testing
- Synchronisations-Overhead
- Cache-Kohärenz und False Sharing
- Skalierung

Deadlock versus Livelock

DEADLOCK (VERKLEMMUNG)

- Zwei oder mehr Threads warten **endlos** auf Ressourcen, die gegenseitig blockiert werden
- **Beispiel**:
  - Thread A hält Lock 1 und wartet auf Lock 2
  - Thread B hält Lock 2 und wartet auf Lock 1
- **Vermeidung**:
  - Immer gleiche Lock-Reihenfolge einhalten
  - Timeouts bei Lock-Anforderungen
  - Vermeiden unnötiger Sperren

LIVELOCK

- Ähnlich Deadlock, aber Threads ändern ständig ihren Zustand in Reaktion aufeinander und kommen dennoch nicht voran
- **Beispiel**:
  - Zwei Prozesse geben immer wieder freiwillig den Lock frei, nehmen ihn aber sofort wieder
- **Vermeidung**:
  - Backoff-Strategien
  - zufällige Wartezeiten

Race Condition versus Starvation

RACE CONDITION (WETTLAUFSITUATION)

- Mehrere Threads greifen gleichzeitig auf eine gemeinsame Ressource zu, Endergebnis hängt von der zeitlichen Ausführung ab
- **Beispiel:**
  - Zwei Threads erhöhen denselben Zähler gleichzeitig → ein Inkrement „geht verloren“
- **Vermeidung:**
  - Synchronisationsmechanismen wie Locks, Mutexes, Semaphore, atomic operations

STARVATION (VERHUNGERN)

- Ein Thread erhält nie CPU-Zeit oder Zugriff auf Ressourcen, weil andere bevorzugt werden
- **Ursache**:
  - Ungerechte Scheduling-Politik
- **Vermeidung**:
  - Faire Locks
  - Prioritätensteuerung

Schwierige Fehlerreproduktion, Debugging und Testing

FEHLERREPRODUKTION

- Fehler hängen oft von exakten Ausführungsreihenfolge ab, die schwer reproduzierbar ist
- **Beispiel:**
  - Race Conditions treten nur unter hoher Last auf
- **Vermeidung:**
  - Logging
  - deterministische Testumgebungen
  - spezielle Debugging-Tools

DEBUGGING & TESTING

- **Probleme**:
  - Non-deterministisches Verhalten
  - Tests müssen Timing-Varianten abdecken
- **Lösungen**:
  - Unit-Tests + Stresstests + Race-Detector-Tools

Cache-Kohärenz und False Sharing

CACHE-KOHÄRENZ

- Mehrere CPU-Cores müssen sich bei gemeinsam genutzten Variablen ständig abstimmen

FALSE SHARING

- Threads schreiben in unterschiedliche Variablen, die aber im gleichen Cache-Line liegen → unnötige Cache-Invalidierungen

Vermeidung:

Daten so anordnen, dass Threads auf getrennte Speicherbereiche zugreifen

Synchronisations-Overhead

- Synchronisationsmechanismen (Locks, Monitore) verursachen Performanceverlust
- **Balance**: Zu viel Synchronisation = langsam; zu wenig = instabil

Amdahlsches Gesetz

Die maximale Beschleunigung ist durch

den sequenziellen Anteil der Aufgabe begrenzt

Übersicht

Algorithmen

- Sortieralgorithmen
- Suchalgorithmen
- Kurze Pfade - Dijkstra

siehe Materialien zur Softwareentwicklung

Übersicht

Software-Engineering

- CASE-Tool
- IDE
- No-Code- und Low-Code-Plattformen
- Prinzipien einer systematischen Programmierung
- Ansätze in der Entwicklung
  - DDD, BDD, TDD etc.
- Automatisierung von Teilaufgaben

CASE-Tool

Computer Aided Software Engineering Tool

- **Computerprogramme, die Softwareentwickelnde unterstützten bei:**
  - Planung
  - Entwicklung
  - Dokumentation
  - Erweiterung/Anpassung von Software
- **automatisierte Funktionen:**
  - Code-Generierung
  - Modellierung
  - Versionsverwaltung

- **Ziel und Zweck:**
  - Produktivität und Qualität des Entwicklungsprozesses steigern
  - Einsatz über den gesamten Lebenszyklus einer Software
  - Strukturierung von Projekten
  - Management von Daten
  - Automatisierung wiederkehrender Aufgaben

IDE

Integrierte Entwicklungsumgebung

- Ermöglicht effizientere Entwicklung

- **Werkzeuge:**
  - Code-Editoren
  - Debugger
  - Build-Automatisierung
  - Version Control Systemen
  - Integrierten Testumgebungen
- **Bsp:** Visual Studio, Eclipse, Intellij

No-Code- und Low-Code-Plattformen

- Entwicklungsumgebungen, die es ermöglichen, Softwareanwendungen weitgehend ohne oder mit nur sehr wenig Programmieraufwand zu erstellen

Kriterium	No-Code	Low-Code
Zielgruppe	Nicht-Programmierende	Entwickelnde & technisch versierte User
Entwicklungsart	Nur visuelle Oberflächen & Drag-and-Drop	Visuell + Möglichkeit, Code einzufügen
Flexibilität	Begrenzt auf Plattformfunktionen	Erweiterbar durch eigenen Code
Beispiele	Wix, Glide, Bubble, Airtable Apps	OutSystems, Mendix, Microsoft Power Apps
Einsatzgebiete	Formulare, einfache Apps, Prototypen	Komplexere Apps, interne Tools, integrierte Systeme
Geschwindigkeit	Sehr schnell bei Standardfunktionen	Schnell, aber mit zusätzlicher Codierung
Abhängigkeit	Hohe Abhängigkeit vom Anbieter	Mittel, da eigener Code möglich
Skalierbarkeit	Eingeschränkt	Besser skalierbar

Prinzipien einer systematischen Programmierung

- Strukturierung
- Modularisierung
- Mehrfachverwendung
- Standardisierung

Strukturierung

- Zerlegung eines Programms in überschaubare, logisch zusammenhängende Abschnitte (Module, Klassen, Funktionen)
- **Ziel:** Bessere Lesbarkeit, Erweiterbarkeit und Testbarkeit
- **Merkmale**
  - Klare Trennung von Daten, Logik und Darstellung
  - Verwendung von Kontrollstrukturen
  - Hierarchische Gliederung des Quellcodes
- **Vorteile**
  - Einfacheres Debugging
  - Schnelleres Verständnis für neue Entwickelnde
  - Weniger Fehler durch klare Strukturen

Modularisierung

- Aufteilung eines Systems in funktional abgeschlossene Einheiten (Module), die unabhängig entwickelt, getestet und erweitert werden können
- Jedes Modul erfüllt klar definierte Aufgabe
- **Merkmale**
  - Lose Kopplung (geringe Abhängigkeiten zwischen Modulen)
  - Hohe Kohäsion (ein Modul hat eine klar umrissene Verantwortlichkeit)
  - Verwendung klar definierter Schnittstellen
- **Vorteile**
  - Kürzere Entwicklungszeit
  - Geringere Fehleranfälligkeit
  - Flexibilität bei Technologieänderungen

Funktionen der Modularisierung

Strukturierung des Programmcodes

Kapselung

- Zugriff nur über definierte Schnittstellen (APIs)

Wiederverwendbarkeit

- Module können in anderen Projekten oder Kontexten erneut verwendet werden

„Wartungs“freundlichkeit

- Änderungen an einem Modul beeinflussen andere Module kaum, sofern die Schnittstelle stabil bleibt
- Fehler lassen sich leichter lokalisieren und beheben

Testbarkeit/Isoliertes Testen

Parallele Entwicklung erhöht Produktivität

Mehrfachverwendung

- Vorhandene Software-Komponenten in mehreren Projekten oder Programmteilen nutzen, statt sie neu zu schreiben
- **Umsetzung**
  - Nutzung von Bibliotheken/Frameworks
  - Entwicklung generischer (universeller) Funktionen oder Klassen
  - Einsatz von Design Patterns
- **Vorteile**
  - Zeit- und Kosteneinsparung
  - Konsistente und bewährte Funktionalität
  - Höhere Qualität, da wiederverwendete Komponenten oft erprobt sind

Bibliotheken und Frameworks

Bibliotheken

- Sammlungen von vorgefertigten Funktionen und Klassen
- Werden von einem Programm verwendet
- Ermöglichen Codeteilung und Code-Wiederverwendung
- Vermeidet Redundanzen

Frameworks

- Strukturierte Sammlungen von Bibliotheken und Tools
- Ermöglichen schnellere Entwicklung komplexer Anwendungen
- Dient als Vorlage einer zu entwickelnden Software

Standardisierung

- Verwendung einheitlicher Vorgaben für Programmierstil, Namenskonventionen, Architekturprinzipien und Technologien
- **Merkmale**
  - Coding Guidelines
  - Einheitliche API-Designs
  - Standardisierte Prozesse für Build, Test, Deployment
  - Verwendung normierter Datenformate (JSON, XML, CSV)
- **Vorteile**
  - Einheitlicher Code erleichtert Teamarbeit
  - Weniger Einarbeitungszeit für neue Entwickelnde
  - Erhöhte Lesbarkeit

Ansätze in der Entwicklung

- Domain-Driven-Design
- Data-Driven-Design
- Behavior-Driven Development (BDD)
- Test-Driven Development (TDD)
- Event-Driven Architecture (EDA)
- Service-Oriented Architecture (SOA)
- Model-Driven Architecture (MDA)
- CRUD-zentrierte Entwicklung

Domain-Driven-Design

- fachliche Anforderungen und die Domäne (also das Geschäfts- bzw. Problemfeld) stehen im Zentrum
- **Kerngedanke**
  - Software soll die Realität der Fachabteilung so genau wie möglich modellieren
  - Entwickelnde und FachexpertInnen arbeiten eng zusammen und sprechen eine gemeinsame Sprache (*Ubiquitous Language*)
  - Fachbegriffe, Prozesse und Regeln der Domäne bestimmen Klassen, Module, Methoden und Datenmodelle

Die Struktur und Sprache der Software sollen

die Struktur und Sprache der Fachdomäne widerspiegeln.

DDD - Zentrale Konzepte

Begriff	Bedeutung
Ubiquitous Language	Einheitliche Sprache zwischen Entwicklung und Fachabteilung Wird im Code, in Dokumentation und Kommunikation verwendet
Bounded Context	Klare Abgrenzung von Teilbereichen (z. B. „Vertrieb“ vs. „Lager“) Jeder Kontext hat eigene Modelle und Begriffe
Entities	Objekte mit eindeutiger Identität (z. B. Kunde, Bestellung).
Value Objects	Objekte ohne eigene Identität, die nur durch ihre Werte definiert sind (z. B. Adresse, Geldbetrag)
Aggregates	Gruppe von Entitäten/Value Objects, die zusammen verwaltet werden (Aggregate Root als Einstiegspunkt)
Repositories	Schnittstellen, um Aggregate aus persistenten Speichern zu laden oder zu speichern
Services	Fachliche Operationen, die nicht direkt zu einer Entität oder einem Value Object gehören
Domain Events	Ereignisse, die eine wichtige Änderung im System widerspiegeln (z. B. BestellungErstellt)

Data-Driven-Design

- Daten und deren Struktur stehen im Vordergrund (nicht die Domäne)
- **Vorgehen:**
  - Erst Datenmodelle und Datenbankstruktur entwerfen, dann Anwendung darum herum bauen
- **Einsatz:**
  - Systeme mit sehr komplexen Datenbeziehungen
  - oft in Data Warehouses oder Reporting-Tools
- **Unterschied zu DDD:**
  - Weniger Augenmerk auf Fachsprache und Domänenlogik
  - mehr auf Tabellen, Felder und Beziehungen

BDD und TDD

BEHAVIOR-DRIVEN DEVELOPMENT

- Fachliches Verhalten und Akzeptanzkriterien vor der Implementierung klar definieren
- **Vorgehen:**
  - Anforderungen in natürlicher Sprache (Given-When-Then) beschreiben, dann automatisiert testen
- **Einsatz:**
  - Projekte mit hoher Bedeutung von automatisierten Akzeptanztests
- **Beispiel (Gherkin-Syntax):**

```
Given ein Kunde mit offenem Warenkorb
When er auf "Bestellen" klickt
Then wird eine Bestellung angelegt
```

TEST-DRIVEN DEVELOPMENT

- Tests schreiben vor dem Code, um Anforderungen schrittweise abzusichern
- **Vorgehen:**
  - **Red**: Test schreiben, der fehlschlägt
  - **Green**: Minimalen Code schreiben, der den Test besteht
  - **Refactor**: Code verbessern
- **Einsatz:**
  - Projekte mit hoher Code-Qualitätsanforderung

EDA – SOA – MDA

EVENT-DRIVEN ARCHITECTURE

- Ereignisse als zentrales Kommunikationsmittel zwischen Komponenten
- **Vorgehen:**
  - Systeme senden und empfangen Events (z. B. BestellungErstellt), um lose Kopplung zu erreichen
- **Einsatz:**
  - Microservices, IoT, Echtzeitanwendungen

SERVICE-ORIENTED ARCHITECTURE

- Große Systeme in eigenständige Services zerlegen, die über standardisierte Schnittstellen kommunizieren
- **Vorgehen:**
  - Prozesse werden in wieder-verwendbare Services aufgeteilt
- **Einsatz:**
  - Unternehmenssoftware
  - Integrationsprojekte

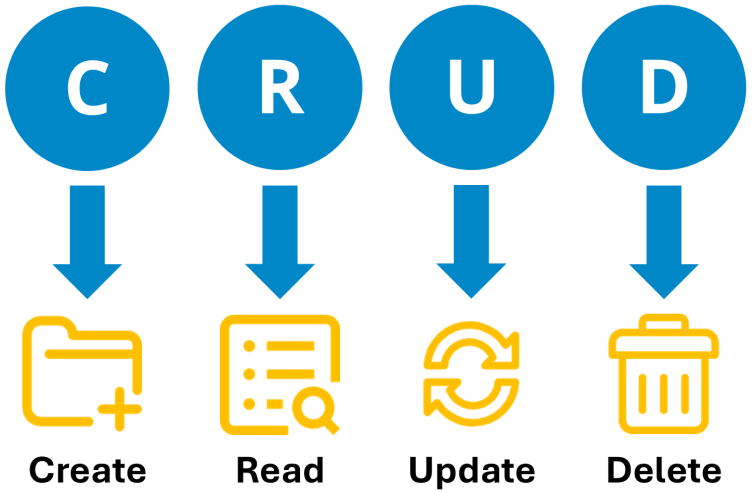
MODEL-DRIVEN ARCHITECTURE

- Zentrale Entwicklung von Modellen, aus denen Code automatisch generiert wird
- **Vorgehen:**
  - Plattformunabhängige Modellierung → Plattformabhängige Umsetzung
- **Einsatz:**
  - Projekte mit hohem Automatisierungsgrad in der Codegenerierung

CRUD-zentrierte Entwicklung

- Umsetzung der typischen Operationen **Create, Read, Update, Delete** auf Daten
- Basis für die meisten datenbankbasierten Anwendungen

- **Einsatz:**
  - Admin-Backends
  - Datenverwaltungssysteme



Automatisierung von Teilaufgaben

Voraussetzungen

- **Stabile, standardisierte Prozesse** mit klaren Regeln und Inputs/Outputs
- **Zugängliche Schnittstellen** (API/CLI/DB/Events) und **ausreichende Rechte**
- **Datenqualität** (IDs, Zustände, Eindeutigkeit)
- **Organisatorische Zustimmung** (Owner, Security, Fachbereich), definierte **SLAs**
- **Technische Basis**: Versionierung (Git), CI/CD, Umgebungen (Dev/Test/Prod)

Wichtige Stolpersteine

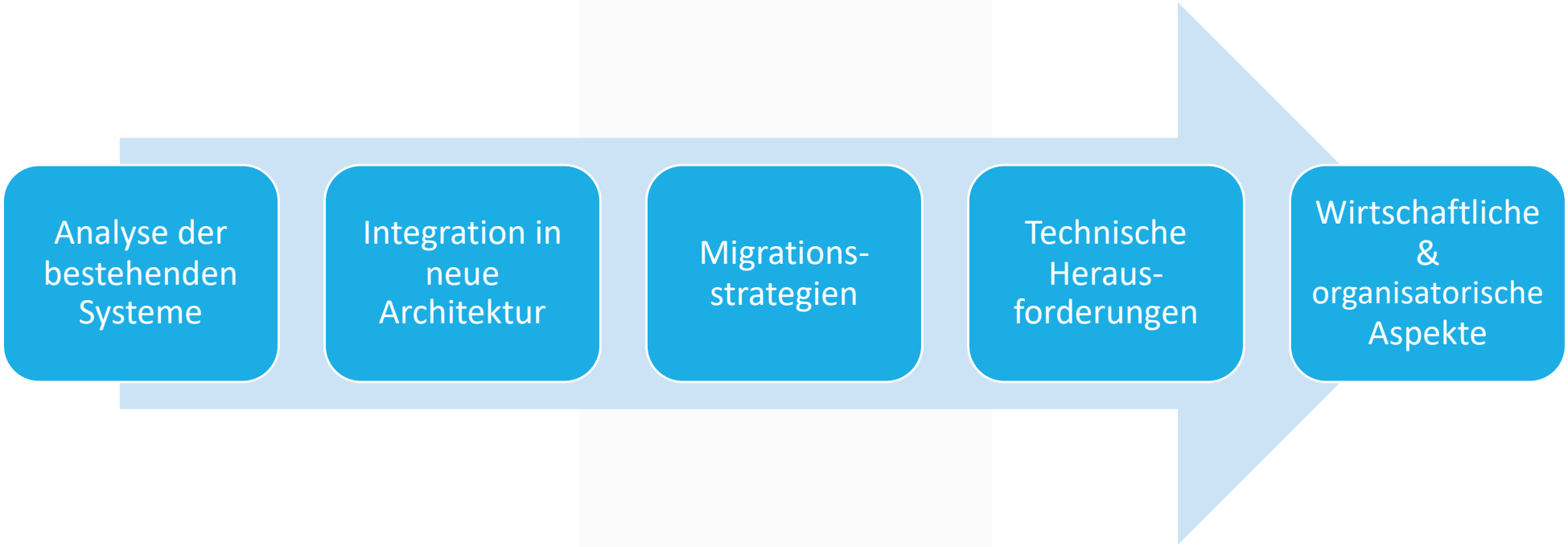
- **UI-Automation bricht schnell** (DOM/Layouts ändern)
- **Über-Automatisierung**: Nur stabile, wertstiftende Teile automatisieren
- **Fehlerkaskaden**: Circuit-Breaker/Dead-Letter-Queues nutzen
- **Sicherheitsrisiken**: Hardcodierte Passwörter, globale Admin-Rechte vermeiden

Übersicht

Softwarearchitektur

- Berücksichtigung bestehender Systeme und Altsysteme
- Modellierungsverfahren
- Softwarearchitektur-Pattern
- Begriffsklärung: Lose Kopplung

Berücksichtigung bestehender Systeme und Altsysteme



Analyse der bestehenden Systeme

Funktionsumfang prüfen:

- Welche Teile werden noch gebraucht, welche sind obsolet?

Technische Basis untersuchen:

- Programmiersprache, Frameworks, Datenbanken, Schnittstellen

Qualität und Stabilität bewerten:

- Wie zuverlässig ist das System, gibt es viele Bugs?

Dokumentationsstand prüfen:

- Oft ist bei Altsystemen wenig oder veraltete Dokumentation vorhanden

Integration in neue Architektur

Schnittstellen nutzen oder neu entwickeln

- z. B. REST-API vor Legacy-Logik schalten

Datenformate anpassen

- z. B. CSV/XML/JSON-Transformation

Kompatibilität sicherstellen

- damit neue und alte Systeme koexistieren können

Zwischenschichten (Adapter, Wrapper) bauen

- um alte Technologien zu kapseln

Migrationsstrategien

Big Bang:

- Altsystem wird komplett durch neues System ersetzt (höheres Risiko)

Schrittweise Migration:

- Teilweise Ersetzung durch modulare Komponenten, oft über Schnittstellenanbindung

Parallelbetrieb:

- Altsystem und Neusystem laufen gleichzeitig, bis das Alte abgeschaltet werden kann

Technische Herausforderungen

Technical Debt

- Veraltete Libraries, unsichere Protokolle, etc.

Leistungsengpässe bei Integration

- z. B. Altsystem hat langsame Batchprozesse

Sicherheitslücken in alten Technologien

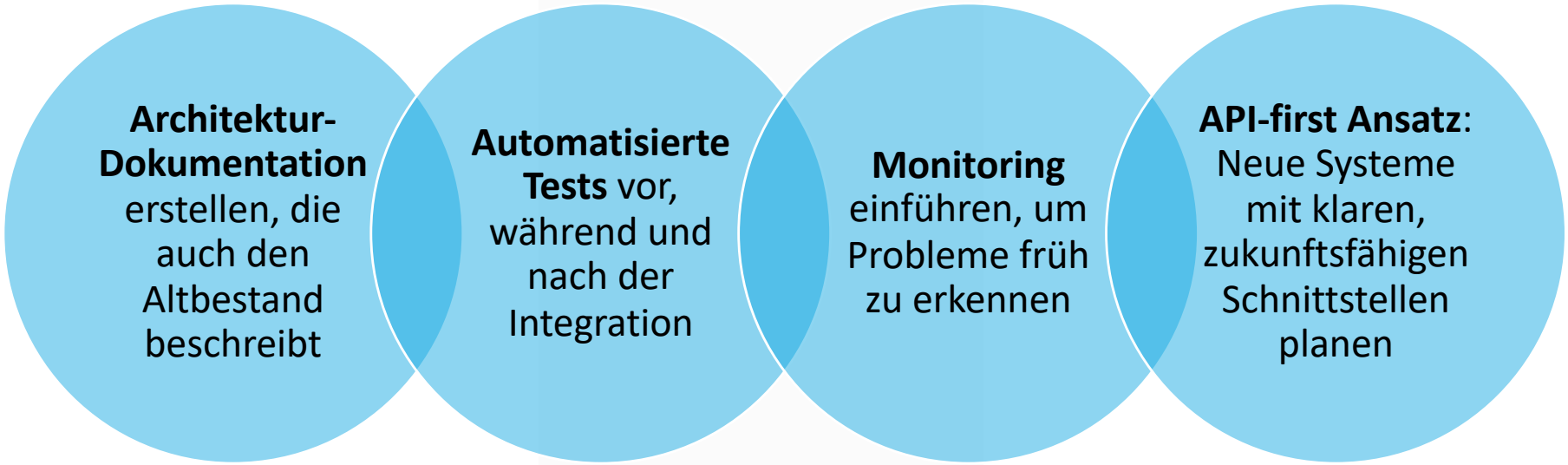
- kein Support, fehlende Patches

Komplexe Abhängigkeiten zu anderen Systemen

Wirtschaftliche & organisatorische Aspekte

Kosten-Nutzen-Abwägung	Risiken minimieren	Schulung
• Lohnt sich Modernisierung oder ist eine komplette Neuentwicklung sinnvoller?	• Datenverlust • Ausfallzeiten • Know-how-Verlust (falls nur wenige Personen das Altsystem kennen	• Entwickelnde und AnwenderInnen müssen ggf. mit der neuen Lösung vertraut gemacht werden

Best Practices



Modellierungsverfahren

TOP-DOWN-VERFAHREN

Vom Allgemeinen zum Speziellen

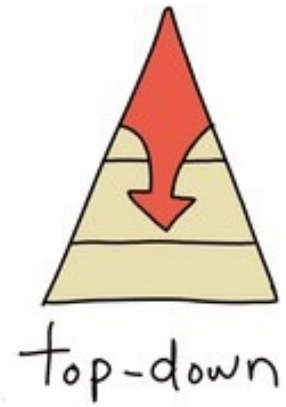
- Man startet mit einer **groben, abstrakten Sicht** auf das Gesamtsystem und verfeinert schrittweise in **konkretere Details**.

BOTTOM-UP-VERFAHREN

Vom Konkreten zum Abstrakten

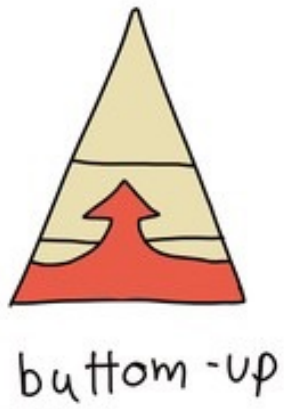
- Man beginnt mit **fertigen oder einfach realisierbaren Bausteinen** und fügt diese zu immer größeren Teilsystemen zusammen

Top-Down-Verfahren



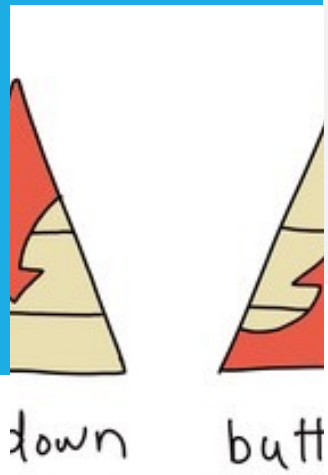
- **Vorgehen:**
  - **Gesamtsystem definieren** – Ziel, Anforderungen, grobe Funktionen festlegen
  - **Grobes Architekturmodell** entwerfen – Hauptkomponenten und deren Schnittstellen
  - **Hierarchische Verfeinerung** – Komponenten werden in Untersysteme und Module zerlegt
  - **Detailentwurf** – Datenstrukturen, Algorithmen, Klassen, konkrete Technologien
- **Vorteile:**
  - Guter Überblick über das gesamte System von Anfang an
  - Anforderungen und Architektur bleiben konsistent
  - Geeignet für komplexe, große Systeme
  - Frühe Erkennung von Widersprüchen zwischen Anforderungen

Bottom-Up-Verfahren



- **Vorgehen:**
  - **Vorhandene Komponenten** (Libraries, Module, Services) identifizieren
  - **Basisfunktionen implementieren** und testen
  - **Komponenten kombinieren**, um größere Funktionalitäten zu erreichen
  - **Gesamtsystem zusammensetzen**
- **Vorteile:**
  - Schnelle Ergebnisse, da früh funktionierende Module entstehen
  - Bessere Wiederverwendung bestehender Softwarebausteine
  - Flexibel bei unklaren oder sich ändernden Anforderungen
  - Gute Grundlage für agile Entwicklung

Hybrider Ansatz



- **Top-Down**
  - für den groben Architekturrahmen
- **Bottom-Up**
  - für die konkrete Umsetzung von Modulen und schnelles Prototyping
- vereint die **Strategieorientierung** des Top-Down mit der **Flexibilität** des Bottom-Up

Softwarearchitektur-Pattern

- Monolith
- 3-Schichten-Modell
- Schichtenmodell
- Microservices
- Pipes and Filters
- Model View Controller
- Model View Presenter
- Model View ViewModel
- Service Oriented Architecture
- REST

Monolithische Architektur

- gesamte Anwendung als eine einzige, in sich geschlossene Codebasis entwickelt, gebaut und ausgeliefert
- Alle Funktionalitäten – Benutzeroberfläche, Geschäftslogik, Datenzugriff – sind eng miteinander verbunden und laufen in einem einzigen Prozess

Merkmale
• Eine Codebasis (Single Codebase) • Ein Build-Prozess • Ein Deployment • Gemeinsame Datenbank • Enge Kopplung

Vorteile
• Einfache Entwicklung & Deployment • Einfache Tests • Hohe Performance intern • Schneller Start für kleine Projekte • Einheitliche Technologie

Schichtenmodell/Layered Architecture

- mehrere hierarchische Ebenen (Layers), jede Schicht kommuniziert nur mit der darüber- und der darunterliegenden Schicht
- **Ziel:**
  - Trennung von Verantwortlichkeiten
  - bessere Strukturierung des Codes

Präsentationsschicht	Anwendungsschicht	Domänenschicht	Datenhaltungsschicht
• Verantwortlich für **Benutzeroberfläche** und Interaktion • Leitet Eingaben an Logikschicht weiter	• Steuert **Ablauf** und **Koordination** von Funktionen • Enthält **Use Cases** und orchestriert Geschäftslogik	• Enthält **Businesslogik** • unabhängig von technischen Details	• Kümmert sich um **technische Details** (Datenbanken, Dateisysteme, Netzwerkzugriffe, externe Services)

3-Schichten-Modell/3-Tier-Architektur

- Spezifische Form des Schichtenmodells, drei klar voneinander abgegrenzte logische Ebenen
- **Ziel:**
  - Verantwortlichkeiten trennen
  - Anwendung flexibler, „wartbarer“ und skalierbarer gestalten

Präsentationsschicht (Presentation Layer)	Logikschicht	Datenhaltungsschicht (Data Layer)
• Zuständig für **Benutzerinteraktion** • Darstellung der Daten (UI) und Entgegennahme von Eingaben	• Enthält die **Geschäftslogik** der Anwendung • Verarbeitet Benutzereingaben, führt Berechnungen durch, validiert Daten • Kommuniziert mit der Datenhaltungsschicht	• Zuständig für **Speichern, Abrufen und Verwalten** von Daten • Besteht meist aus einer Datenbank oder anderen Speichersystemen

Vor- und Nachteile einer Schichtenarchitektur

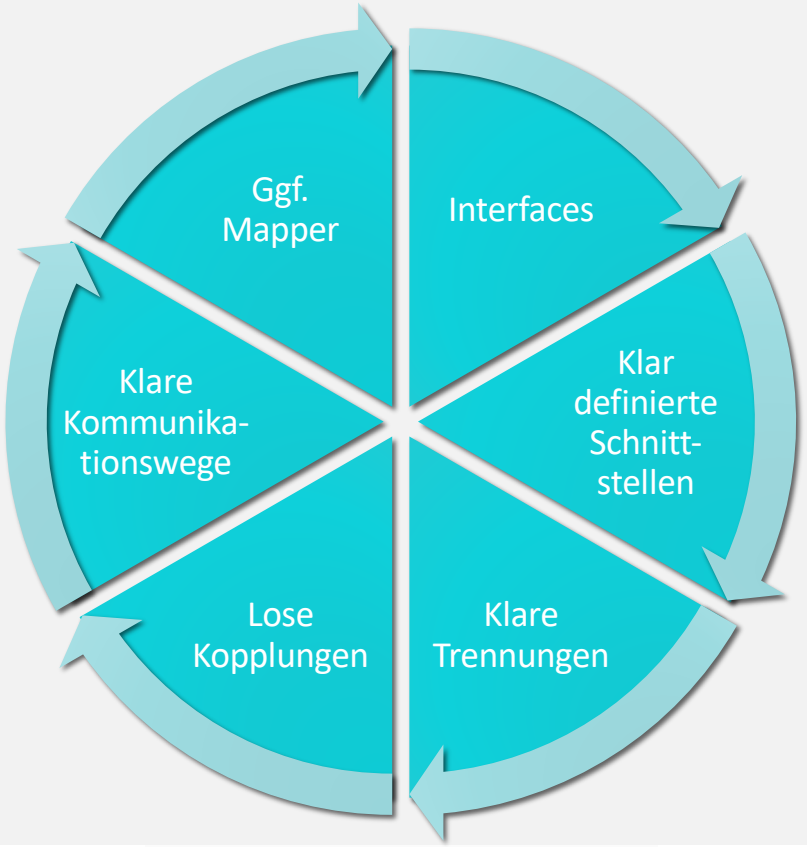
VORTEILE

- Klare Verantwortlichkeiten
- Einfachere Erweiterbarkeit
- Wiederverwendbarkeit
- Testbarkeit
- Technologieunabhängigkeit

NACHTEILE

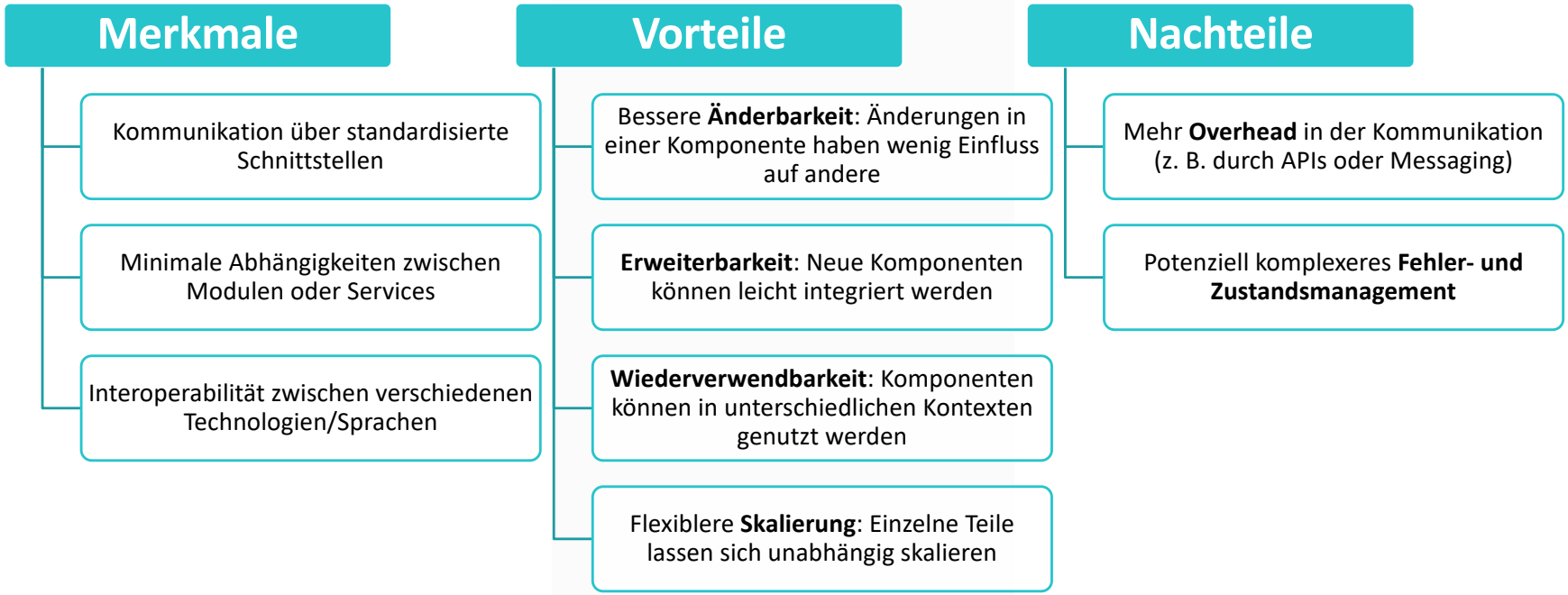
- Zusätzlicher Overhead (Planungsaufwand, Code, ggf. Performance)
- Höhere Komplexität
- Strikte Schichtendisziplin erforderlich

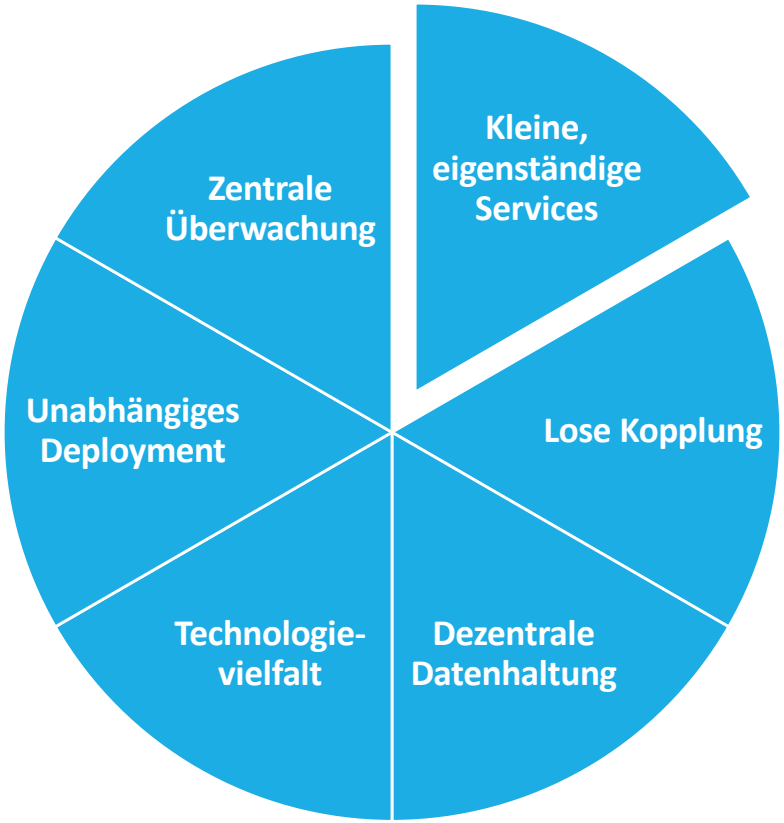
Typische Merkmale einer Schichten-Architektur



Begriffsklärung: Lose Kopplung

- Komponenten einer Softwarearchitektur arbeiten möglichst unabhängig voneinander
- Änderungen in einer Komponente erfordern nur minimale oder keine Änderungen in anderen Komponenten





Microservices

- Sammlung kleiner, unabhängiger Dienste (Services), die
  - jeweils eine klar abgegrenzte Geschäftsaufgabe erfüllen und
  - über standardisierte Schnittstellen (meist HTTP/REST oder Messaging) miteinander kommunizieren
  - unabhängig entwickelt, getestet, deployed und skaliert werden

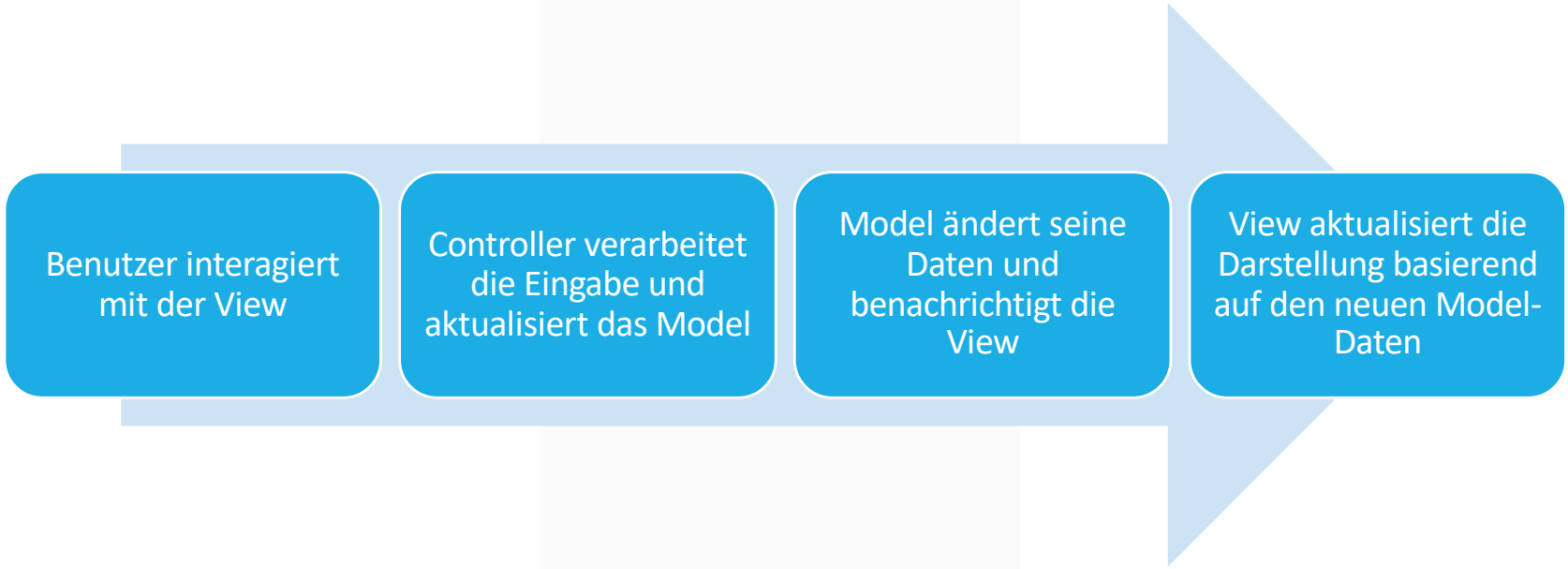
- **Vorteile:**
  - Bessere Skalierbarkeit
  - Technologische Freiheit
  - Hohe Erweiterbarkeit
  - Unabhängige Entwicklung
  - Fehlertoleranz
  - Schnelleres Deployment

Model-View-Controller (MVC)

- Trennung von Daten, Darstellung und Steuerung einer Anwendung
- Unterstützt parallele Entwicklung

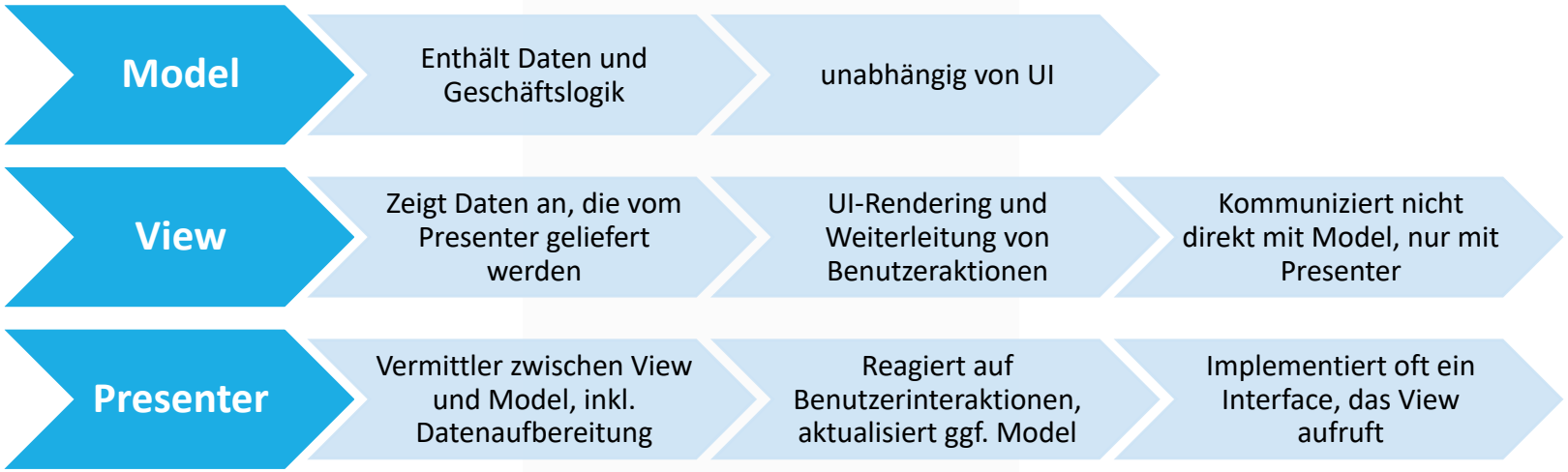
Model	Enthält Daten und Geschäftslogik	unabhängig von Darstellung und Eingabe	Reagiert auf Änderungen der Daten, informiert View
View	Präsentiert die Daten aus dem Model in geeigneter Form (UI)	Zeigt keine Geschäftslogik, sondern nur Darstellung	Kann mehrere Views für dasselbe Model geben
Controller	Nimmt Benutzereingaben entgegen	Interpretiert diese und verändert Model oder wählt eine View	Vermittler zwischen Benutzer, Model und View

MVC - Ablauf

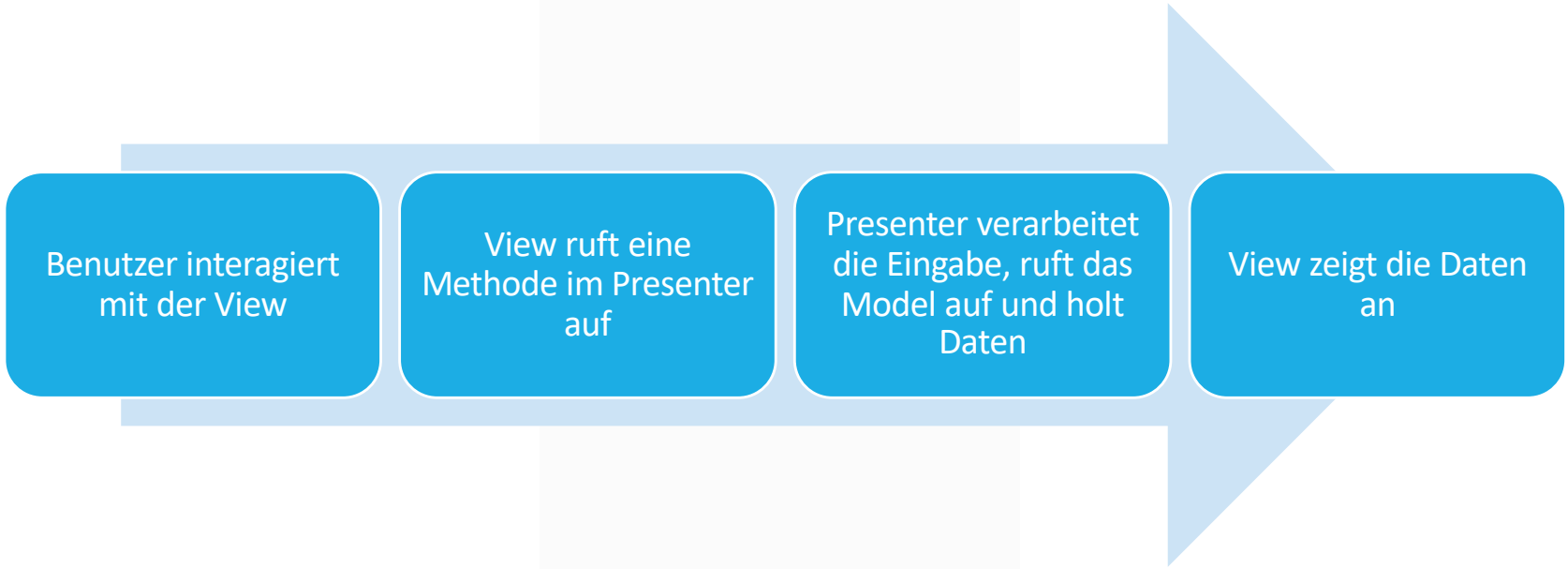

Benutzer interagiert mit der View
Controller verarbeitet die Eingabe und aktualisiert das Model
Model ändert seine Daten und benachrichtigt die View
View aktualisiert die Darstellung basierend auf den neuen Model-Daten

Model View Presenter (MVP)

- aus MVC hervorgegangen mit konsequenterer Trennung zwischen Darstellung und Logik
- Präsentationslogik in eigene Komponente (Presenter) ausgelagert
- Die View ist passiv – zeigt nur an, was Presenter vorgibt

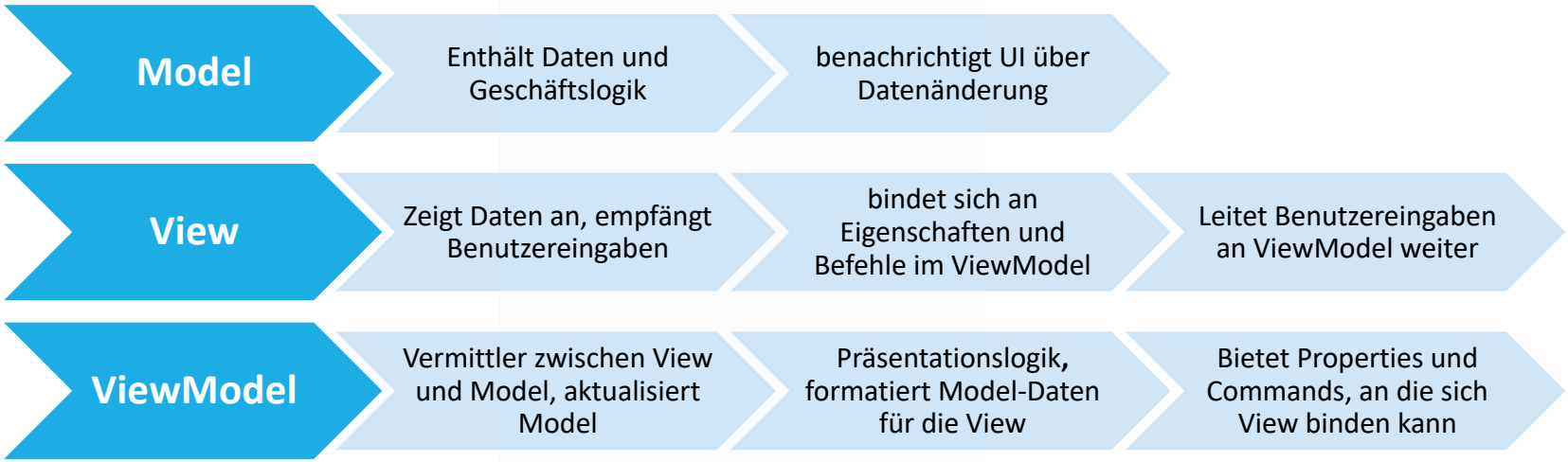


MVP - Ablauf

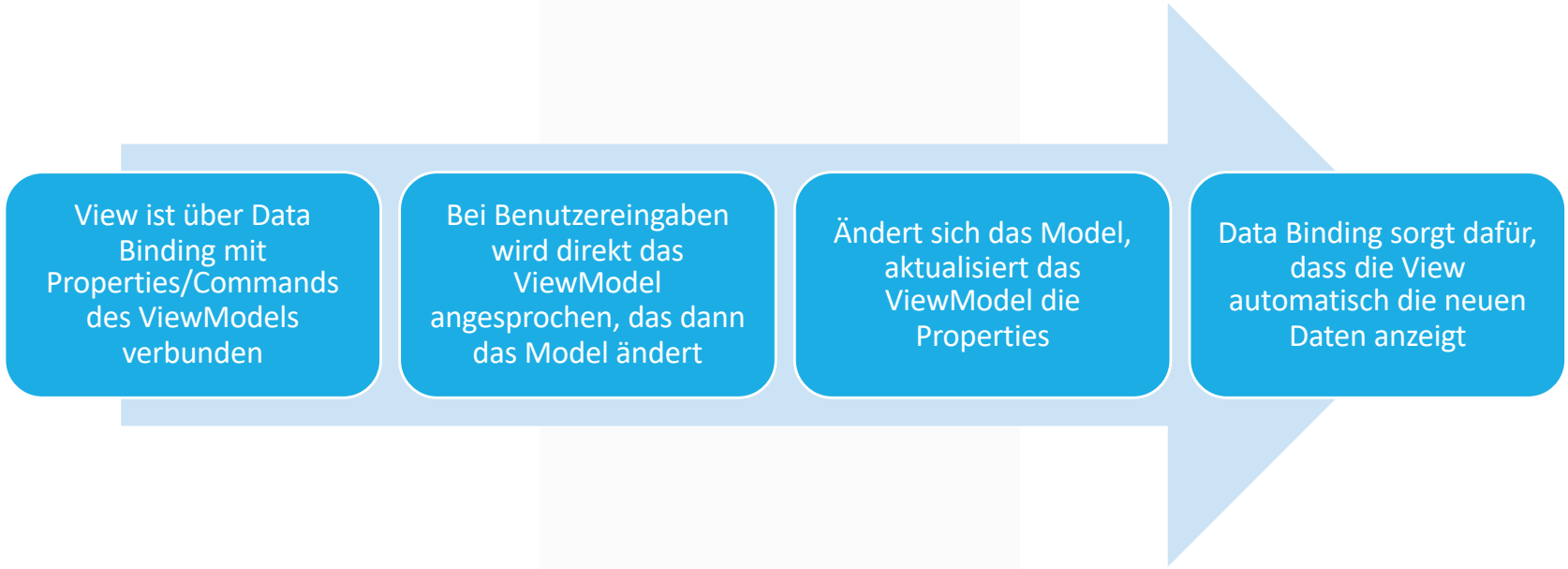

Benutzer interagiert mit der View
View ruft eine Methode im Presenter auf
Presenter verarbeitet die Eingabe, ruft das Model auf und holt Daten
View zeigt die Daten an

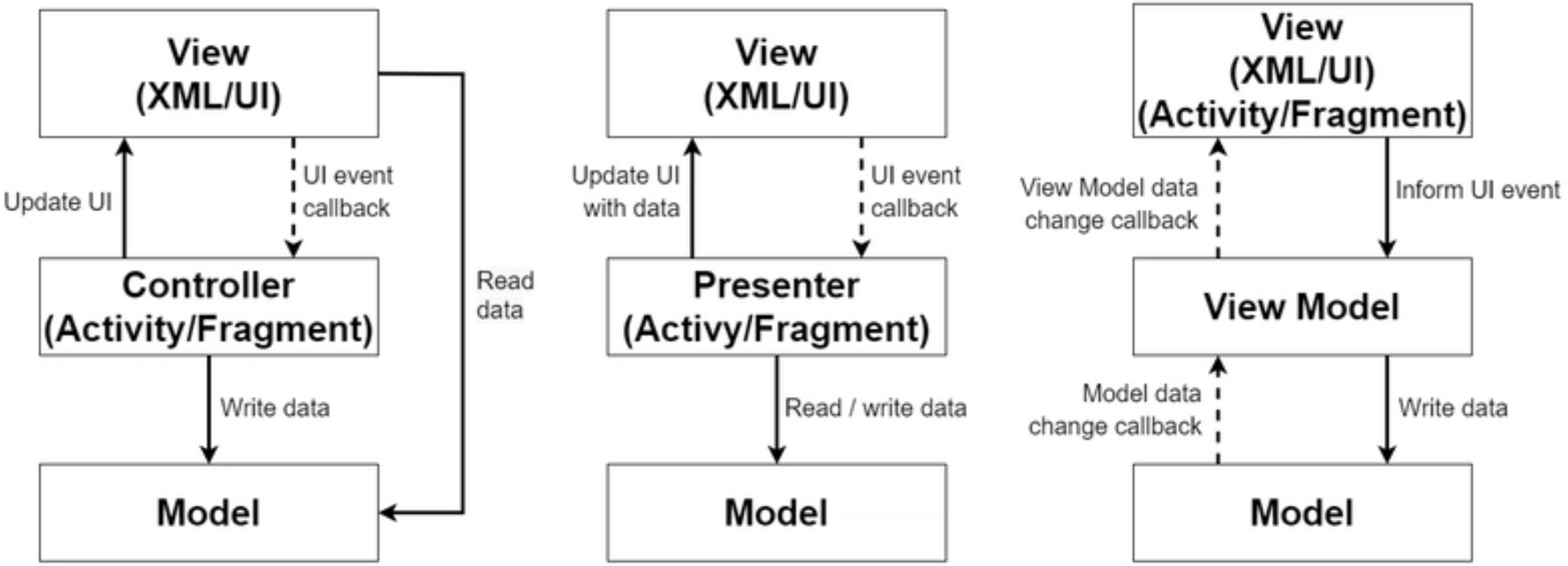
Model View ViewModel (MVVM)

- trennt Darstellung, Daten und Präsentationslogik
- nutzt automatisches Binding, um View und ViewModel zu synchronisieren – ohne dass die View direkte Logik enthält



MVVM - Ablauf


View ist über Data Binding mit Properties/Commands des ViewModels verbunden
Bei Benutzereingaben wird direkt das ViewModel angesprochen, das dann das Model ändert
Ändert sich das Model, aktualisiert das ViewModel die Properties
Data Binding sorgt dafür, dass die View automatisch die neuen Daten anzeigt



MVC versus MVP versus MVVM

Pipes and Filters 1/2

- Daten fließen in einem Verarbeitungsschritt-für-Schritt-Prozess durch eine Reihe von Filtern
- Zwischen Filter werden Daten durch Pipes (Verbindungen) weitergereicht

FILTER	PIPES
▪ Führt klar abgegrenzte Verarbeitung der Daten durch ▪ unabhängig von anderen Filtern ▪ Nimmt Eingaben entgegen und gibt verarbeitete Ausgaben weiter	▪ Stellen Datenfluss zwischen Filtern her ▪ Können in-memory, als Streams oder als Netzwerkverbindungen realisiert sein

Pipes and Filters 2/2

MERKMALE

- **Lose Kopplung**
  - Jeder Filter kennt nur Eingabe- und Ausgabedaten
- **Einfache Erweiterbarkeit**
  - Neue Filter können leicht hinzugefügt oder bestehende ausgetauscht werden
- **Parallelisierung möglich**
  - Mehrere Filter können gleichzeitig arbeiten
- **Streaming-fähig**
  - Daten können verarbeitet werden, während sie noch ankommen

BEST PRACTICES

- **Zustandslose Filter**
  - jeder Aufruf sollte das gleiche Ergebnis liefern
- **Standardisierte Datenformate** zwischen Filtern
- **Logging und Monitoring in den Pipes**
  - um Fehlerquellen zu finden
- **Performance messen**
  - viele Übergänge erzeugen schnell Overhead
- **Parallele Verarbeitung**
  - z. B. bei großen Datenmengen

Übersicht

Schnittstellen

- Datenaustausch zwischen Systemen
- Webservices
- SOA
- Datenaustauschformate
  - CSV
  - JSON
  - XML (inkl. DTD, Schema)
- Weitere Datenaustausch- und Markup-Formate
- Vergleich von Datenaustausch- und Markup-Formaten
- REST (inkl. HATEOAS)
- SOAP
- WSDL

Datenaustausch zwischen Systemen

Datei-basierter Datenaustausch	Systeme exportieren Daten in Dateien, die von anderen Systemen importiert werden (CSV, XML, JSON)
API-basierter Austausch	Systeme kommunizieren direkt über definierte Schnittstellen (REST; SOAP)
Datenbank-basierter Austausch	Gemeinsame Nutzung einer Datenbank oder Replikation zwischen Datenbanken
Messaging- & Event-basierter Austausch	Systeme senden Ereignisse oder Nachrichten, die andere Systeme konsumieren (z. B. Event-Bus in Microservices)
Remote Procedure Calls (RPC)	Anwendung ruft Funktionen/Methoden in einer anderen Anwendung auf, als ob sie lokal wäre
Manuelle & halbautomatische Methoden	Für kleine Datenmengen oder einmalige Übertragungen (z. B. Export/Import über Excel)

Webservices

- Schnittstelle, die auf offenen Protokollen wie HTTP, SOAP oder REST basiert und es ermöglicht, dass Systeme unabhängig von ihrer Technologie miteinander interagieren
- ermöglichen es Anwendungen, Daten oder Funktionen plattformübergreifend auszutauschen

SOAP (Simple Object Access Protocol)	REST (Representational State Transfer)	GraphQL (modernere Alternative)
• Nutzt XML als Datenformat • Formal, geeignet für B2B-Kommunikation • Unterstützt Sicherheit, Transaktionen, Verlässlichkeit	• Nutzt HTTP direkt (GET, POST, PUT, DELETE …) • Datenformate: JSON, XML • Leichtgewichtig, gut für Web & Mobile • Einfach zu implementieren und zu testen	• Abfragesprache für APIs • Client bestimmt, welche Daten er braucht

SOA

Service-Oriented Architecture

- Architekturkonzept, bei dem Softwarefunktionen als Services bereitgestellt werden, die über standardisierte Schnittstellen kommunizieren

Service Provider (stellt den Service bereit, inkl. Logik und Daten)

Service Registry (Verzeichnis aller verfügbaren Services)

Service Consumer (ruft Service über Schnittstelle auf)

-> Komponenten

- **Wichtigste Merkmale:**
  - Dienste sind **lose gekoppelt** (unabhängig voneinander)
  - Jeder Dienst erfüllt eine **klar definierte Aufgabe**
  - Kommunikation erfolgt über **offene Standards** (z. B. HTTP, XML)
  - Ideal für **verteilte, heterogene Systeme** (z. B. Java ↔ .NET ↔ Python)
  - **Plattformunabhängigkeit, Wiederverwendbarkeit, Erweiterbarkeit**

Datenaustauschformate

Merkmal	CSV	XML	JSON
Formattyp	Flach/tabellarisch	Hierarchisch (markup)	Hierarchisch (objektbasiert)
Lesbarkeit	Einfach (auch manuell)	Komplexer, aber lesbar	Sehr gut lesbar
Strukturierbarkeit	Gering	Hoch	Hoch
Datentypen	Nicht typisiert	Möglich (mit XSD)	Typisiert
Standardisierung	Kaum	Hoch (W3C)	Hoch (RFC 8259)
Einsatzbereiche	Datenbanken, Tabellen	Webservices, Konfig-Dateien	REST-APIs, Web-Apps
Speicherplatzbedarf	Gering	Hoch	Mittel
Tool-Unterstützung	Sehr hoch	Sehr hoch	Sehr hoch

CSV

Comma-Separated Values

- einfaches Textformat, in dem Daten zeilenweise gespeichert und die einzelnen Werte durch ein Trennzeichen (meist Komma oder Semikolon) getrennt werden
- Sehr einfach zu lesen und erzeugen
- Häufig genutzt für Tabellenexporte
- Kein offizieller Standard: Trennzeichen können variieren
- Keine Verschachtelung/Nesting möglich
- Keine Typisierung (alle Daten sind Strings)

Typische Struktur:

```
Name,Alter,Stadt
Anna,28,Berlin
Lukas,35,Hamburg
```

JSON

JavaScript Object Notation

- leichtgewichtiges, objektbasiertes Austauschformat, das besonders im Webbereich (REST-APIs) verwendet wird
- gut lesbar
- Unterstützt komplexe Strukturen (Arrays, Objekte, verschachtelte Daten)
- Ideal für moderne Web-APIs (REST)
- Unterstützt primitive Datentypen: Zahl, String, Boolean, null
- Standard in JavaScript-basierten Anwendungen
- Kleinere Datenmenge im Vergleich zu XML bei gleicher Information

Typische Struktur:

```
{
  "name": "Anna",
  "alter": 28,
  "stadt": "Berlin"
}
```

JSON - Einsatzgebiete

Webentwicklung	REST-APIs, JavaScript Frontends
Mobile Apps	Client-Server-Kommunikation
Microservices	Service-Kommunikation, Event-Streams
Konfiguration	Einstellungsdateien für Tools & Frameworks
Testautomatisierung	Testdaten, Mock-Daten, Validierungs-JSON
Datenaustausch	Export/Import, Middleware, Schnittstellen
Datenvisualisierung	Input für Charts & Dashboards
Machine Learning / Big Data	JSON als Datenformat für Analyse und Modellinput/-output

JSON – Vor- und Nachteile

VORTEILE

- Einfach & leichtgewichtig
- Menschlich lesbar
- Sprachunabhängig
- Strukturierte Daten
- Sehr gut für APIs
- Direkt nutzbar in JavaScript
- Effizient in der Übertragung
- Einfach zu parsen

NACHTEILE

- Keine Kommentare erlaubt
- Keine Datentypenprüfung
- Kein Standard für Referenzen
- Kein Namespacing
- Wenig Metainformationen
- Nur UTF-8 (keine Binärdaten)
- Verschachtelung fehleranfällig

XML

eXtensible Markup Language

- hierarchisches Textformat zur Darstellung strukturierter Daten
- verwendet Tags, um Daten und ihre Struktur zu definieren
- Selbstbeschreibend: Datenstruktur ist im Text enthalten
- Unterstützt Verschachtelungen
- Plattformunabhängig, maschinen- und menschenlesbar
- Häufig in SOAP-Webservices, Konfigurationsdateien
- Umfangreicher als CSV, daher größere Datenmenge
- Tools: XSD (Schema-Validierung), XPath, XSLT

Typische Struktur:

```
<person>
  <name>Anna</name>
  <alter>28</alter>
  <stadt>Berlin</stadt>
</person>
```

XML - Wohlgeformtheit und Validität

WOHLGEFORMTHEIT

- grundlegenden Regeln der XML-Syntax werden eingehalten:
  - genau ein Wurzelelement (Root-Element)
  - Alle Elemente korrekt geöffnet und geschlossen
  - Tags sind verschachtelt, dürfen sich nicht überlappen
  - Attributwerte in Anführungszeichen
  - Groß- und Kleinschreibung ist relevant
  - Keine ungültigen Zeichen

VALIDITÄT

- Wohlgeformt und zusätzlich den Regeln einer definierten Struktur entspricht:
  - Dokument referenziert ein Schema oder eine DTD
  - Alle Elemente und Attribute entsprechen den definierten Datentypen, Reihenfolgen und Vorkommen

XML – Parser und Serialisierer

PARSER

- Softwaremodul, das ein XML-Dokument einliest, analysiert und es in eine verarbeitbare Datenstruktur (z. B. DOM-Objektbaum) überführt
- **Aufgaben:**
  - Prüfen auf Wohlgeformtheit
  - Optional: Prüfen auf Validität (DTD oder XML Schema)
  - Konvertieren des XML-Dokuments in eine baumartige Datenstruktur, die von Programmen genutzt werden kann
- **DOM-** Liest das gesamte Dokument in den Speicher und baut einen Objektbaum auf
- **SAX-** Ereignisbasierter Parser – verarbeitet das Dokument sequenziell beim Einlesen

SERIALISIERER

- wandelt Datenstrukturen oder Objekte in ein XML-Dokument um, sodass es gespeichert, übertragen oder weiterverarbeitet werden kann
- **Aufgaben:**
  - Konvertieren von Objekten, Arrays oder Strukturen in gültiges XML
  - Sicherstellen, dass die erzeugte XML wohlgeformt ist
  - Optional: Strukturkonformität sicherstellen (Validität)
- **Beispiele:**
  - XMLWriter
  - ElementTree

DTD

Document Type Definition

- beschreibt, welche Elemente und Attribute in einem XML-Dokument erlaubt sind, in welcher Reihenfolge sie auftreten dürfen und wie sie verschachtelt werden
- **Zweck:**
  - Validierung von XML-Dokumenten
  - Sicherstellen, dass ein XML-Dokument einer erwarteten Struktur entspricht
  - Erleichtert die Datenüberprüfung, Interpretation und den Datenaustausch

```
<!DOCTYPE person [
  <!ELEMENT person (name, alter)>
  <!ELEMENT name (#PCDATA)>
  <!ELEMENT alter (#PCDATA)>
]>
<person>
  <name>Anna</name>
  <alter>28</alter>
</person>
```

Schema (XSD)

- Weiterentwicklung und Erweiterung der DTD
- Zusätzlich zu DTD: Pflichtfelder, Namensräume, Einschränkungen, komplexe Datentypen
- **Zweck:**
  - Präzisere Validierung von XML-Dokumenten auf Struktur & Datentypen
  - Sicherstellung eines einheitlichen Datenformats
  - Automatisierte Verarbeitung durch Programme (z. B. Webservices)
  - Dokumentation der erwarteten XML-Struktur

```xml
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="person">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="name" type="xs:string"/>
        <xs:element name="alter" type="xs:integer"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Weitere Schema-Sprachen für XML

RELAX NG (RELAX NEXT GENERATION)

- Einfachheit, Klarheit und Flexibilität
- **Vorteile:**
  - Besonders in der Compact Syntax sehr übersichtlich und lesbar
  - Unterstützung von Datentypen wie bei XSD
  - Teile von Schemas können wiederverwendet werden
  - Kann strukturierte oder unstrukturierte Inhalte definieren

SCHEMATRON

- regelbasiert, erlaubt inhaltliche Prüfregeln in Form von XPath-Ausdrücken
- **Merkmale:**
  - Regeln werden mit Bedingungen (Tests) beschrieben
  - XPath-Ausdrücke ermöglichen gezielte Prüfungen auf Inhalt & Kontext
  - Kann auch logische Abhängigkeiten oder Kontextbedingungen prüfen
  - Ideal für komplexe Prüfungen

XML Transformation

XSLT – EXTENSIBLE STYLESHEET LANGUAGE TRANSFORMATIONS

- Transformation von XML-Daten in z. B.:
  - HTML
  - Text
  - ein anderes XML-Format
  - CSV oder JSON
- beschreibt mit einer Sammlung von Regeln (Templates), wie ein XML-Dokument umgewandelt werden soll

XSL-FO – EXTENSIBLE STYLESHEET LANGUAGE FORMATTING OBJECTS

- formatgetreuen Darstellung von XML-Inhalten, insbesondere für:
  - PDF-Erzeugung
  - Drucklayouts
  - Dokumentation mit exaktem Layout
- Umwandlung mittels XSLT in XSL-FO-Daten
- anschließend mit einem Renderer in z. B. PDF überführt

Weitere Datenaustausch- und Markup-Formate

SGML

- **Standard Generalized Markup Language**
- Mutter aller Auszeichnungs-sprachen
- Sehr komplex – wird heute kaum noch direkt verwendet

HTML

- **Hypertext Markup Language**
- Spezialisierter XML/SGML-Ableger
- Fokus: Darstellung von Inhalten im Browser

YAML

- **Yet Another Markup Language**
- Unterstützt Kommentare, Mehrzeilige Strings, verschachtelte Daten
- Achtung: Einrückung entscheidend – Fehleranfällig!

TOML

- **Tom's Obvious, Minimal Language**
- Beliebt für moderne Konfigurations-dateien
- Besser strukturierbar als INI, aber einfacher als YAML

INI

- **Initialiserung**
- Uralt, sehr einfach, aber keine komplexen Datenstrukturen

Vergleich von Datenaustausch- und Markup-Formaten

Format	Typ	Zweck / Einsatzgebiet	Strukturierung	Lesbarkeit	Datentypen	Kommentare	Schema-Validierung
SGML	Meta-Auszeichnung	Basis für XML/HTML, technisch sehr mächtig	Sehr komplex	Nein	Frei definierbar	Ja	Ja
XML	Markup	Datenstrukturierung, Konfiguration, Webservices	Strukturiert	Mittel	Tags, Attribute	Nein	DTD, XSD, RelaxNG
HTML	Markup	Darstellung von Webinhalten	Dokumentbasiert	Gut	Text, Tags	Nein	Teilweise (DTD)
JSON	Datenformat	Web-APIs, moderne Datenübertragung	Strukturiert	Gut	Zahlen, Text, Array, Objekte	Nein	Teilweise (JSON-Schema)
CSV	Tabellendaten	Tabellen, Ex-/Import von Daten (Excel, DB)	Flach	Sehr gut	Nur Strings	Nein	Nein
YAML	Datenformat	Konfigurationsdateien, lesbarer als JSON	Strukturiert	Sehr gut	Listen, Objekte	Ja	Teilweise (Schema)
TOML	Konfiguration	Konfigurationsdateien (z. B. für Python-Projekte)	Strukturiert	Sehr gut	Zahlen, Bool, Listen	Ja	Nein
INI	Konfiguration	Einfache Key-Value-Konfiguration	Einfach	Sehr gut	Strings	Ja	Nein

REST

Representational State Transfer

- Architekturstil für verteilte Systeme, insbesondere für Webservices basierend auf HTTP
- beschreibt, wie Ressourcen (z. B. Datenobjekte wie „Benutzer“, „Artikel“) über das Internet adressiert und manipuliert werden können, meist über die HTTP-Methoden
- Definiert Kommunikation zwischen Clients und Server
- setzt auf eine ressourcenorientierte URL-Struktur
- ist zustandslos (stateless)
- verwendet häufig JSON als Austauschformat

HTTP-Methode	Bedeutung (CRUD)	Beispiel
GET	Lesen	/users/123 → Gibt Benutzer zurück
POST	Erstellen	/users + JSON-Daten
PUT	Aktualisieren	/users/123 + neue Daten
DELETE	Löschen	/users/123

REST – Vor- und Nachteile

VORTEILE

Vorteil	Erklärung
Einfachheit	Nutzt bestehende HTTP-Standards
Plattform-unabhängig	Funktioniert mit jedem HTTP-fähigen Client
Hohe Skalierbarkeit	Durch Zustandslosigkeit leicht verteilbar
Flexibel in der Darstellung	JSON, XML oder andere Formate möglich
Klare Struktur	Ressourcenorientierter Ansatz ist leicht verständlich

NACHTEILE

Herausforderung	Erklärung
Kein Standard für komplexe Transaktionen	Mehrere Änderungen in einem Schritt schwer umsetzbar
HATEOAS wird selten vollständig implementiert	Viele REST-APIs verzichten auf Hypermedia
Sicherheits-anforderungen	Zugriffskontrolle und Authentifizierung müssen separat implementiert werden
Overhead bei sehr einfachen Anwendungen	Für kleine APIs kann REST unnötig komplex wirken

REST - Adressierbarkeit

- jede Ressource (z. B. ein Benutzer, Artikel, Produkt) ist über eindeutige URL ansprechbar

`GET /users/123` ⟶ **holt Benutzer mit ID 123**

`DELETE /products/456` ⟶ **löscht Produkt mit ID 456**

- **Nutzen:**
  - Klare, nachvollziehbare Struktur
  - Links zwischen Ressourcen sind möglich (HATEOAS)
  - Macht Webservices „verlinkbar“ und leicht testbar

REST - Zustandslosigkeit

- Server speichert keinen Zustand über den Client zwischen zwei Anfragen
- Jede HTTP-Anfrage muss alle notwendigen Informationen enthalten, damit der Server sie ausführen kann

Ein Client sendet bei jeder Anfrage ein Authentifizierungstoken (z. B. Bearer xyz123), weil der Server nicht „weiß“, wer gerade angemeldet ist.

- **Vorteile:**
  - Horizontale Skalierbarkeit (Server können einfach repliziert werden)
  - Einfachere Fehlersuche
  - Keine Server-Session notwendig

REST – Einheitliche Schnittstelle

- alle Ressourcen werden auf dieselbe Weise angesprochen und manipuliert, unabhängig von ihrer Art

HTTP-Header + Body erklären, was übertragen wird

z. B. Content-Type: application/json

- **Vorteile:**
  - Trennung von Client und Server
  - Einheitliches API-Design → einfacher zu verstehen, zu verwenden und zu dokumentieren

REST – Ressource versus Repräsentation

RESSOURCE

- abstrakter Begriff für ein eindeutig identifizierbares Objekt, Konzept oder Datenelement im System
- **Beispiele:**
  - Ein Benutzer (/users/123)
  - Ein Blogbeitrag (/posts/42)
  - Eine Bestellung (/orders/99)
- **Eigenschaften:**
  - Wird über eine URI angesprochen (z. B. GET /users/123)
  - Ist nicht gleich dem Inhalt, sondern eher der „Ort“ oder die „Idee“ hinter den Daten

REPRÄSENTATION

- konkrete Darstellung (z. B. in XML, JSON, HTML) der Ressource, die über das Netzwerk übertragen wird
- **Beispiel:**

```
{
  "id": 123,
  "name": "Max Mustermann"
}
```

- **Eigenschaften:**
  - Wird vom Server an den Client gesendet
  - Enthält den aktuellen Zustand der Ressource
  - Format ist verhandelbar über HTTP-Header (Accept: application/json)

Hypermedia as the Engine of Application State (HATEOAS)

- REST-Client hat alle Informationen, die für die Navigation durch die Anwendung notwendig sind
- erhält diese dynamisch vom Server über sogenannte Hypermedia-Links
- nicht fest eincodiert
- Vorteile:
  - Entkoppelt Client & Server: Clients müssen keine URIs hartkodieren
  - Flexibilität: Der Server kann Links dynamisch ändern
  - Selbstdokumentierende APIs

```
{
  "id": 123,
  "name": "Max Mustermann",
  "_links": {
    "self": { "href": "/users/123" },
    "orders": { "href": "/users/123/orders" },
    "edit": { "href": "/users/123/edit" }
  }
}
```

REST – Code on Demand

- optionales REST-Prinzip
- Server sendet Client bei Bedarf ausführbaren Code, z. B. in Form von:
  - JavaScript (in Webbrowsern)
  - Skripten, die vom Client dynamisch ausgeführt werden können

- **Ziel:**
  - Ermöglicht dem Client eine flexiblere Funktionalität, ohne dass dieser vorher alle Funktionen selbst kennen oder implementieren muss

SOAP

Simple Object Access Protocol

- XML-basiertes Protokoll zur strukturierten Kommunikation zwischen Anwendungen über ein Netzwerk
- **Merkmale:**
  - Plattform- und sprachunabhängig
  - Nutzt üblicherweise HTTP oder SMTP als Transportprotokoll
  - Definiert ein standardisiertes Nachrichtenformat (Request/Response)

Unterstützt:

- Fehlermeldungen
- Sicherheit (z. B. WS-Security)
- Transaktionen
- Verlässlichkeit

WSDL

Web Services Description Language

- XML-basierte Beschreibungssprache, mit der ein (zumeist SOAP-basierender) Webservice formal beschrieben wird

- **Zweck:**
  - Maschinenlesbare Definition des Servicevertrags

Enthält Informationen über:

- Verfügbare Methoden (Operationen)
- Datenformate (Input/Output)
- Protokolle und Endpunkte

Übersicht

Containerisierung

- Container und Docker
- Begriffsklärung: Orchestrierung
- Kubernetes

Container und Docker

- **Container**: leichtgewichtige, portable Pakete, die folgendes beinhalten:
  - Anwendungen
  - alle notwendigen Abhängigkeiten (Bibliotheken, Konfigurationen, Laufzeitumgebung)
- **Docker**: bekannteste Plattform zur Erstellung, Bereitstellung und Ausführung von Containern

Eigenschaften

- **Isolation**: Jeder Container läuft in eigener Umgebung
- **Portabilität**: Container-Image läuft identisch auf jedem System mit Docker
- **Schneller Start**: Container starten in Sekunden, da kein vollständiges OS gebootet wird
- **Weniger Overhead** als VMs: Nutzen den Kernel des Hostsystems
- **Imageschichten (Layer)**: Änderungen werden in Layern gespeichert (Copy-on-Write)

Verwendungs-zwecke

- **Anwendungsdeployment**:
  - „Write once, run anywhere“
- **Microservices**:
  - Jeder Service läuft in eigenem Container
- **CI/CD**:
  - Einheitliche Build- und Testumgebungen
- **Skalierung**:
  - Container lassen sich schnell vervielfachen
- **Legacy-Apps isolieren**:
  - Alte Anwendungen in isolierter Umgebung betreiben

Worauf ist zu achten

- **Security**: Minimale Base-Images nutzen, keine sensiblen Daten ins Image legen
- **Imagegröße** klein halten für schnelleren Download & Start
- **Versions-Tagging** verwenden, nicht nur latest
- **Ressourcenlimits** setzen (CPU, RAM)
- **Persistenz**: Daten in Volumes statt im Container speichern
- **Monitoring & Logging** integrieren

Begriffsklärung: Orchestrierung

- Koordination, Verteilung und Überwachung von Containern in einem Cluster
- stellt sicher, dass:
  - immer die richtige Anzahl von Containern läuft
  - diese untereinander vernetzt sind
  - Last gleichmäßig verteilt wird

Ohne Orchestrierung müsste man:

- ✓Container **manuell starten** und stoppen
- ✓Ausfälle **händisch beheben**
- ✓Updates **manuell ausrollen**
- ✓Netzwerk- und Service-Adressen **selbst verwalten**
- ➢bei **mehreren Containern oder Microservices** praktisch unmöglich
- ➢Orchestrierungssysteme automatisieren diese Prozesse

Hauptaufgaben der Orchestrierung

Aufgabe	Beschreibung
Deployment	Container automatisch starten und auf Knoten im Cluster verteilen
Skalierung	Anzahl der Container nach Last automatisch erhöhen oder verringern (Auto-Scaling)
Lastverteilung (Load Balancing)	Eingehende Anfragen gleichmäßig auf Container verteilen
Self-Healing	Fehlerhafte Container neu starten oder auf andere Nodes verschieben
Service Discovery	Container automatisch auffindbar machen (Namensauflösung, interne DNS)
Ressourcenmanagement	CPU- und RAM-Limits setzen, faire Verteilung
Netzwerkmanagement	Sichere und zuverlässige Kommunikation zwischen Containern bereitstellen
Rolling Updates / Rollbacks	Neue Versionen schrittweise ausrollen und bei Problemen zurückrollen
Monitoring & Logging	Status, Logs und Metriken zentral erfassen

Kubernetes

- **Orchestrierungssystem** für Container
- Automatisiert **Deployment**, **Skalierung**, **Lastverteilung**, **Selbstheilung** und **Management** von Containern in einem Cluster

Eigenschaften

- **Cluster-Architektur**: Ein Master-Node steuert Worker-Nodes
- **Pods**: Kleinste Kubernetes-Einheit (ein oder mehrere Container, die gemeinsam Ressourcen nutzen)
- **Services**: Stellen Netzwerkschnittstellen für Pods bereit
- **Deployments**: Beschreiben gewünschte Anzahl und Version von Pods
- **Autoscaling**: Skaliert Pods automatisch nach Last
- **Self-Healing**: Neustart fehlerhafter Container, Neuzuweisung bei Node-Ausfall
- **Rolling Updates / Rollbacks**: Versionen schrittweise ausrollen und bei Problemen zurückrollen

Verwendungs-zwecke

- **Microservices-Orchestrierung**
- **Skalierbare Webanwendungen**
- **Multi-Cloud- oder Hybrid-Umgebungen**
- **Batch- und Hintergrundjobs**
- **CI/CD-Pipelines**
  - mit automatischer Bereitstellung

Worauf ist zu achten

- **Konfiguration als Code**
- **Namespaces** für Mandantentrennung und Ordnung
- **Ressourcenlimits & Quotas**, um Überlastung zu vermeiden
- **Service Mesh** für Sicherheit, Routing, Observability
- **Monitoring** und **Logging**
- **Security**: Secrets-Management, Netzwerk-Policies
- **Kostenkontrolle** bei Cloud-K8s-Clustern

Übersicht

App-Entwicklung

- Herausforderungen plattformübergreifender mobiler Entwicklung
- Überblick
  - Native App
  - Hybrid App
  - Cross-Platform App
  - Responsive Web App

Herausforderungen plattformübergreifender mobiler Entwicklung

- Entwickelnde müssen mehrere Technologien beherrschen oder Frameworks nutzen, die plattformübergreifend arbeiten
- Unterschiedliche Plattform-APIs
- Unterschiedliche UI-/UX-Richtlinien
- Performance-Unterschiede zwischen Native Apps und Hybriden Web-Apps
- Test- und Aufwand bei Änderungen/Anpassungen/Erweiterungen
- Unterschiedliche Store-Richtlinien, Freigabeprozesse und ggf. Store-spezifische Funktionen
- Verschiedene Sicherheitsrichtlinien, Verschlüsselungen, Zertifikatsmanagement, Sandbox-Verhalten

App-Entwicklung

Merkmal	Native App	Hybrid App	Cross-Platform App	Responsive Web	Progressive Web App (PWA)
Performance	⭐⭐⭐⭐⭐	⭐⭐⭐	⭐⭐⭐⭐	⭐⭐	⭐⭐⭐ bis ⭐⭐⭐⭐
Zugriff auf Hardware	Voll	Teilweise (via Plugins)	Meist voll	Sehr eingeschränkt	Eingeschränkt (abhängig vom Browser)
UI/UX	Beste	Gut	Sehr gut	Mittel	Gut (app-ähnlich)
Offline-Fähigkeit	Ja	Ja	Ja	Nur mit PWA-Ansatz	Ja (Service Worker)
Veröffentlichung	App-Store	App-Store	App-Store	Browser	Browser / Installierbar
Entwicklungskosten	Hoch	Mittel	Mittel	Gering	Gering bis Mittel
Plattform-unabhängigkeit	Nein	Ja	Ja	Ja	Ja
Updates	Manuell über App-Store	Manuell über App-Store	Manuell über App-Store	Automatisch beim Laden	Automatisch im Hintergrund

Native App

- App wird für ein bestimmtes Betriebssystem (iOS, Android, Windows) mit den plattform-eigenen Programmiersprachen und Tools entwickelt

- **Technologien:**
  - iOS: Swift, Objective-C, Xcode
  - Android: Kotlin, Java, Android Studio

Vorteile:	Nachteile:
• Maximale Performance • Zugriff auf alle Hardware- und OS-Funktionen • Beste User Experience (UI/UX)	• Entwicklung für jede Plattform separat → höhere Kosten • Erweiterung/Anpassung aufwendiger

Hybrid App

- App besteht aus Webtechnologien (HTML, CSS, JavaScript)
- läuft in einer WebView innerhalb einer nativen Hülle

- **Technologien:**
  - Ionic, Apache Cordova, Capacitor

Vorteile:	Nachteile:
○ Einmal entwickeln, überall ausführen ○ Kostengünstiger als native Entwicklung ○ Zugriff auf einige native Funktionen via Plugins	○ Meist langsamer als nativ ○ Eingeschränkte UI-Performance bei komplexen Animationen

Cross-Platform App

- Gemeinsamer Code wird für mehrere Plattformen kompiliert
- teils mit nativen UI-Elementen

- **Technologien:**
  - Flutter, React Native, Xamarin

Vorteile:	Nachteile:
○ Gemeinsamer Code für iOS & Android ○ Nahezu native Performance (v. a. bei Flutter, React Native) ○ Gute Balance aus Entwicklungszeit und Performance	○ Manche native Funktionen müssen plattformspezifisch nachimplementiert werden ○ Abhängigkeit von Framework-Updates

Responsive Web App

- Webanwendung mit responsivem Design
- passt sich Bildschirmgrößen an (Smartphone, Tablet, Desktop)

- **Technologien:**
  - HTML5, CSS3 (Flexbox, Grid), JavaScript
  - Frameworks wie Angular, React, Vue

Vorteile:	Nachteile:
○ Eine Anwendung für alle Geräte mit Browser ○ Keine App-Store-Zulassung nötig ○ Günstig und schnell umsetzbar	○ Kein Offline-Betrieb (außer als PWA) ○ Eingeschränkter Zugriff auf Hardwarefunktionen ○ Abhängig von Browser-Performance

Übersicht

Pseudocode und UML

- Pseudocode
- Aktivitätsdiagramm
- Use Case Diagram
- Klassendiagramm
- Zustandsdiagramm
- Sequenzdiagramm

Pseudocode

- Dient der meist einfachen Darstellung von Algorithmen und Programmen unabhängig einer Programmiersprache
- kann sehr unterschiedlich aussehen, siehe Bsp. im Zusatzdokument
- Hier wird **EINE** Möglichkeit aufgeführt

Verzweigungen

```
if <Bedingung>
   <eingerückte Anweisungen im If-Teil>*
[else
   <eingerückte Anweisung im Else-Teil>*]
```

Schleifen

```
repeat
   <eingerückte Anweisung>*
until <Endebedingung>
```

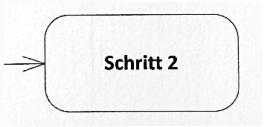
```
while <Bedingung>
   <eingerückte Anweisung>*
```

```
for <Initialisierung> to|downto <Endebedingung> [by <delta>]
   <eingerückte Anweisung>*
```

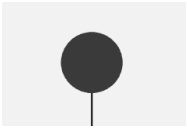
Aktivitätsdiagramm 1/4

- Aktivität (Rechteck mit abgerundeten Ecken)
  - Ablaufreihenfolge von Aktionen
  - Bildet eine Art Rahmen um alle Aktionen einer Aktivität

- Anmerkung/Kommentar
  - Kommentare zu Elementen



- Aktion
  - ausführbaren Teilbereiche
  - Früher: Aktivität



- Startknoten
  - Startpunkt oder den Anfangszustand einer Aktivität darzustellen
  - kann allein stehen oder durch ein Notizsymbol mit erklärendem Kommentar erweitert werden

Aktivitätsdiagramm 2/4

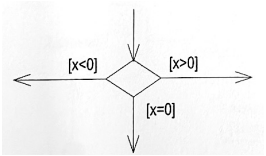
- Kontrollfluss
  - Kontrollfluss von einer Aktion zur anderen

- Objektfluss
  - Weg von Objekten

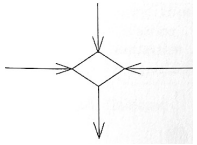
- Endknoten
  - Ende aller Kontrollflüsse innerhalb der Aktivität
  - Abschluss sämtlicher Prozessabläufe

- Ablaufende
  - Ende eines einzelnen Kontrollflusses

Aktivitätsdiagramm 3/4



- Entscheidung
  - bedingten Verzweigungspunkt mit einem Eingang und mehreren Ausgängen



- Zusammenführung (**Oder**)
  - Zusammenfließen von Strömen

- Teilung
  - Aktivitätsfluss, der sich in zwei oder mehr parallele Ströme verzweigen kann

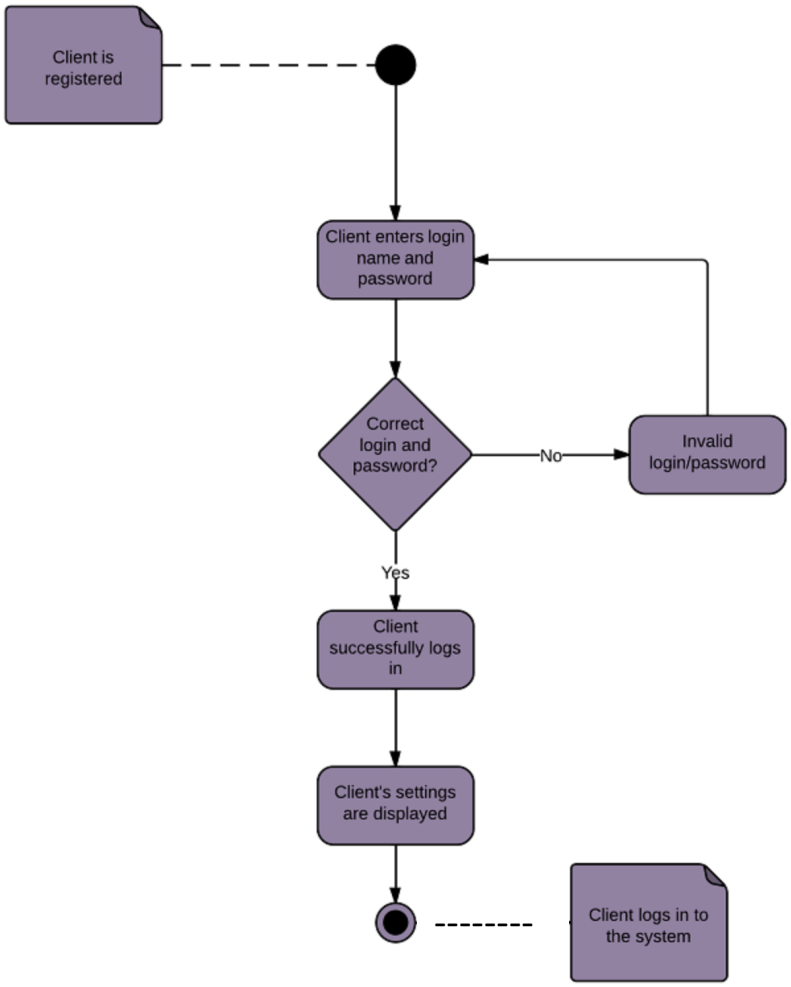
- Synchronisation (**Und**)
  - Verbindung zwischen mehreren gleichzeitig laufenden Aktivitäten, die in einen Aktivitätsstrom führen

Aktivitätsdiagramm 4/4

- Senden von Signalen
  - Senden eines Signals an eine annehmende Aktivität

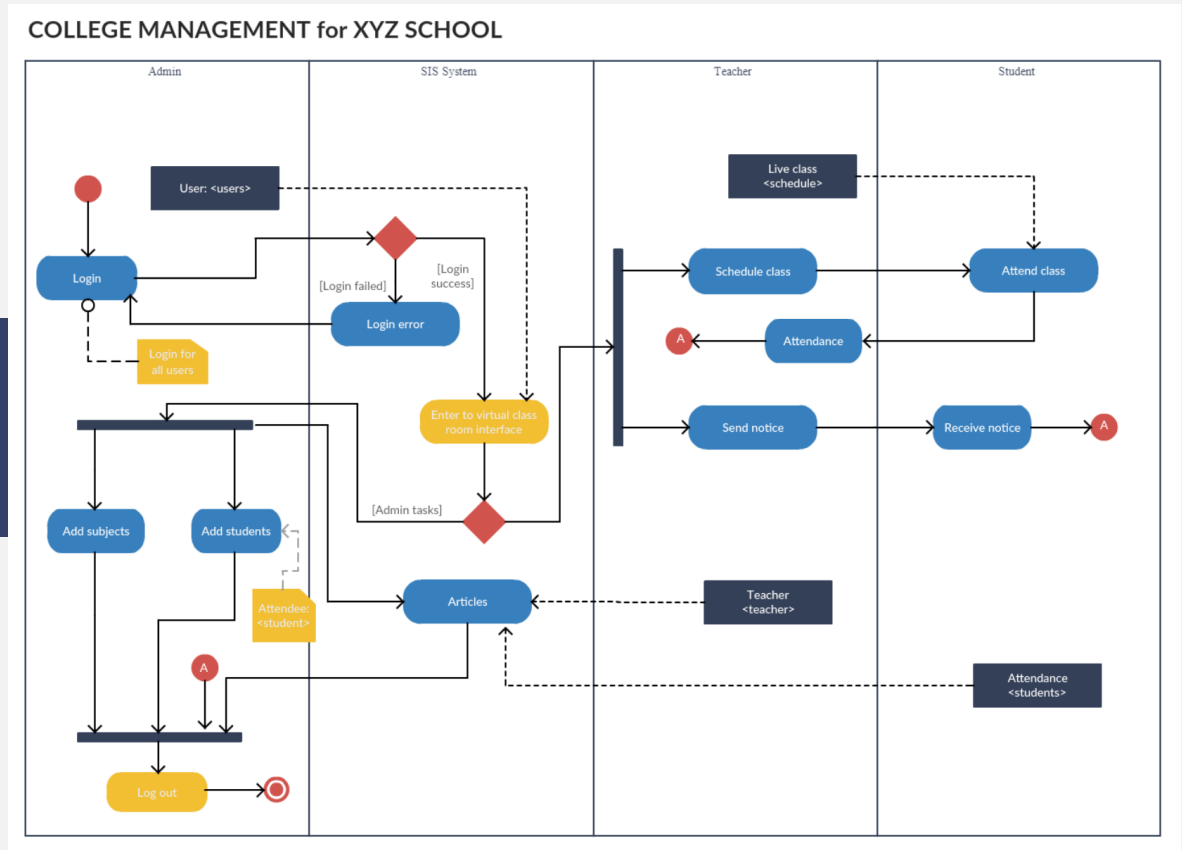
- Empfang von Signalen
  - Signal wird empfangen

Aktivitäts-diagramm Beispiel 1



Quelle

Aktivitäts-diagramm Beispiel 2

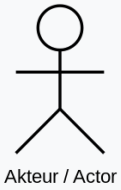


Quelle

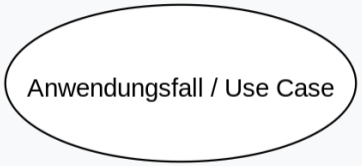
Use Case Diagram 1/3

- Elemente

Der **Systemkontext** wird durch **Systemgrenzen** in Form von Rechtecken gekennzeichnet.



Akteure werden als „Strichmännchen“ dargestellt, welche sowohl Personen wie Kunden oder Administratoren als auch ein System darstellen können.

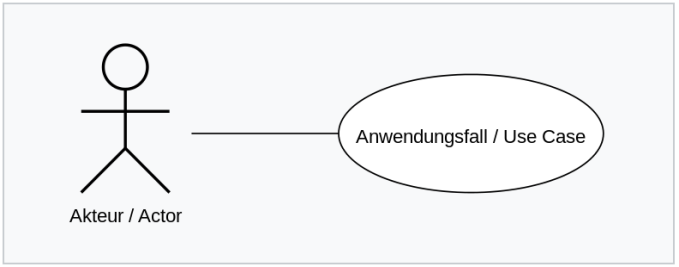


Anwendungsfälle werden in Ellipsen dargestellt. Sie müssen (z. B. in einem Kommentar oder einer eigenen Datei) beschrieben werden.

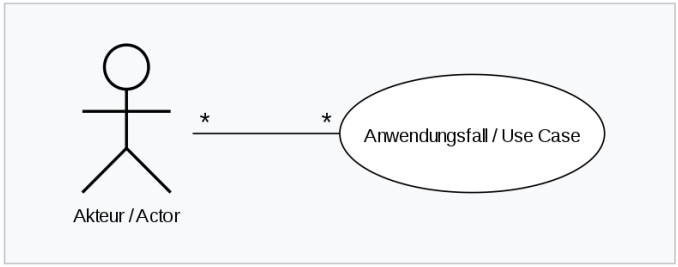
Quelle

Use Case Diagram 2/3

- Beziehungen Teil 1



Assoziation / Kommunikation von Akteur und Anwendungsfall.

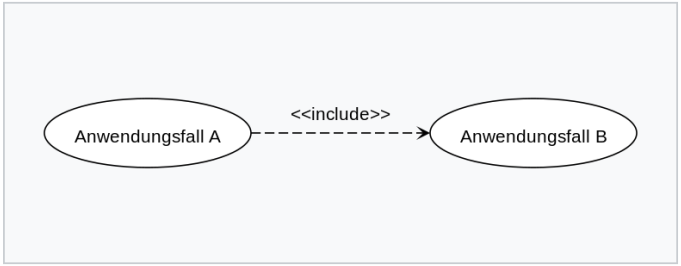


Multiplizität von Akteur und Anwendungsfall, wobei die Voreinstellung des Akteurs 1 ist.

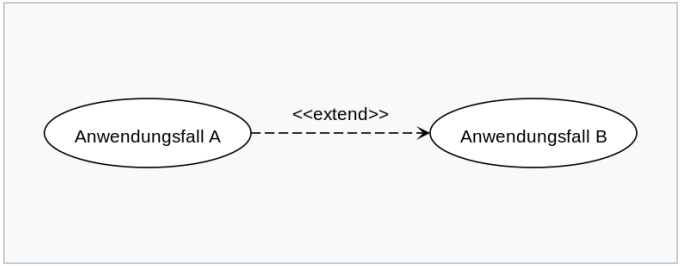
Quelle

Use Case Diagram 3/3

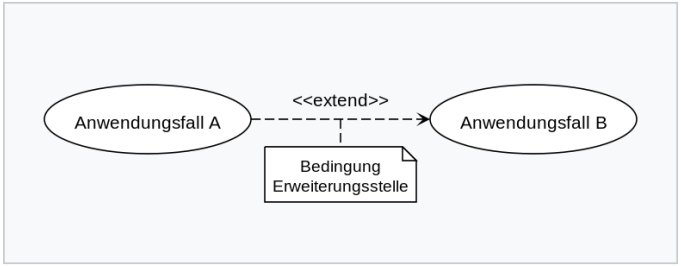
- Beziehungen Teil 2



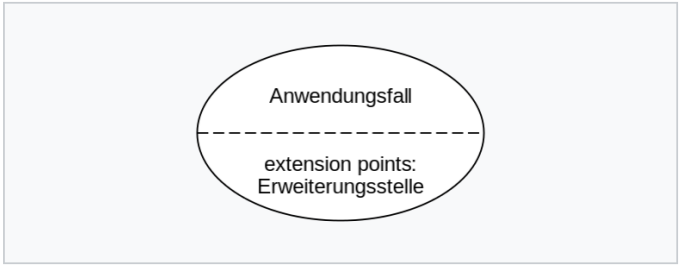
Include-Beziehung im Anwendungsfalldiagramm, wobei Anwendungsfall A den Anwendungsfall B beinhaltet.



Extend-Beziehung im Anwendungsfalldiagramm, wobei Anwendungsfall A den Anwendungsfall B erweitert.



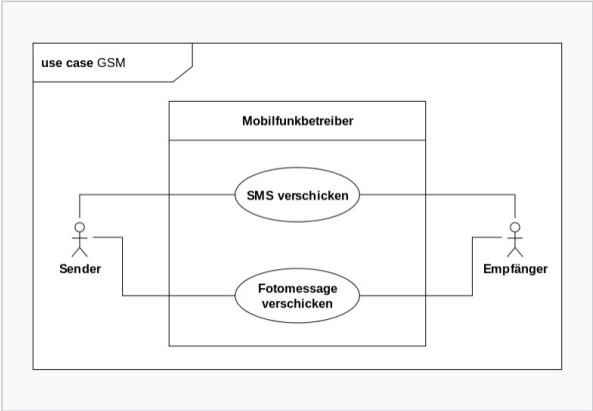
Extend-Beziehung mit extension point, wobei Anwendungsfall A den Anwendungsfall B unter der angegebenen Bedingung erweitert.



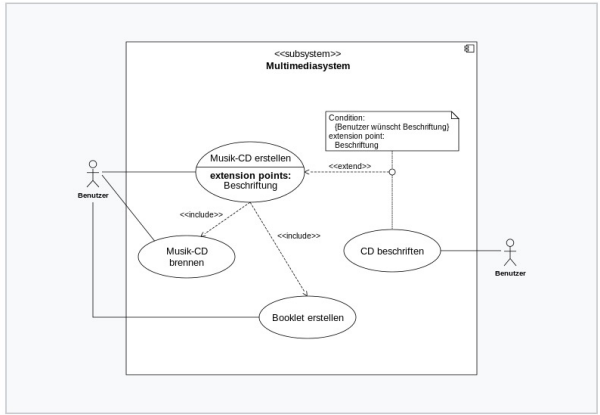
Anwendungsfall mit extension point.

Quelle

Use Case Diagram Beispiel



Das Schlüsselwort im Kopfbereich ist *use case*. Das Anwendungsfalldiagramm in der Abbildung links ist in einen Kopf- und in einen Inhaltsbereich getrennt und mit einem Rahmen umschlossen, so wie es die UML2 neu für alle Diagramme vorsieht.

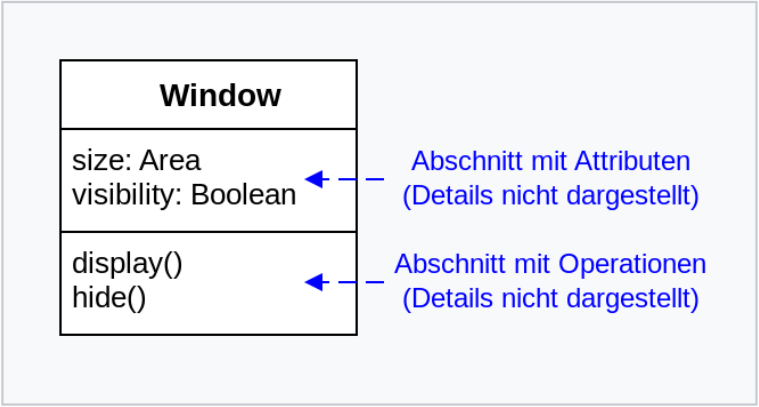


Ein komplexeres Anwendungsfalldiagramm, das die Beziehungen zwischen dem Akteur `Benutzer` und dem System `Multimediasystem` festhält. Ein Benutzer ist an vier Anwendungsfällen interessiert, die ihrerseits untereinander in Beziehung stehen. `Musik-CD erstellen` ist der komplexeste Anwendungsfall, weil er zwei andere Anwendungsfälle importiert und optional durch einen dritten, `CD beschriften`, erweitert wird.

Quelle

Klassendiagramm - Klassen

- Beispiel:
  - **Klasse** „Window“
  - **Attribute**: size und visibility
  - **Methoden**: display(), hide()

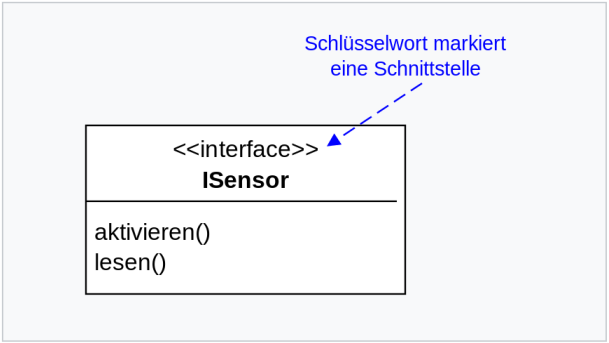


Quelle

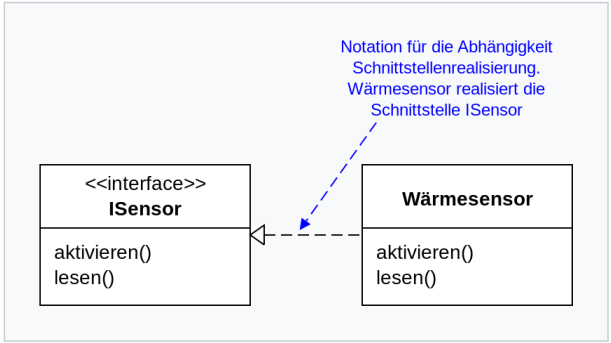
Klassendiagramm - Interfaces

Beispiel:

- Interface „ISensor“
- Methoden: aktivieren(), lesen()
- Klasse „Wärmesensor“ leitet von „ISensor“ ab

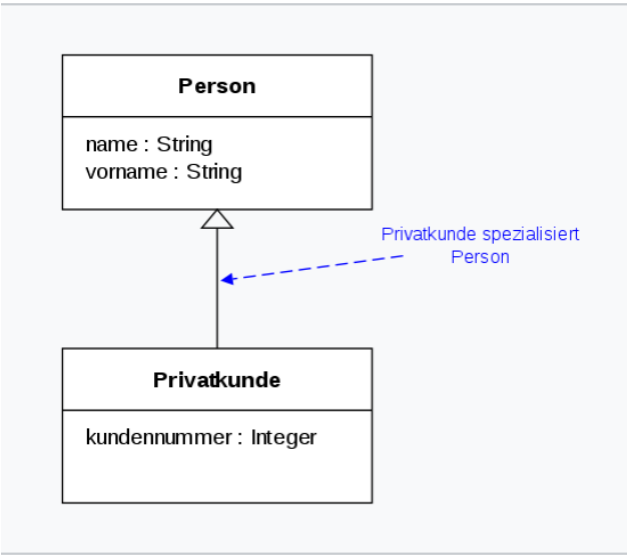


Quelle



Klassendiagramm - Generalisierung

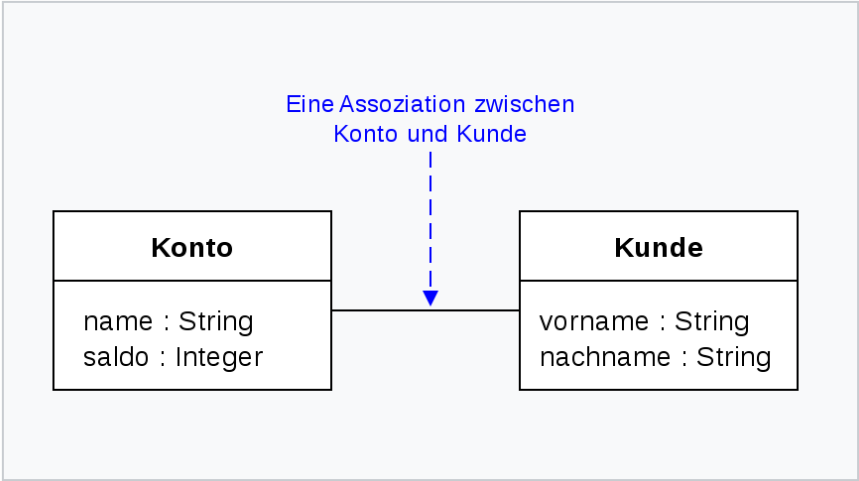
- Gerichtete Beziehung zwischen einer generelleren und einer spezielleren Klasse
- Exemplare der spezielleren Klasse sind Exemplare der generelleren Klasse
- Speziellere Klasse verfügt implizit über alle Merkmale (Struktur- und Verhaltensmerkmale) der generelleren Klasse



Quelle

Klassendiagramm - Assoziation

- Beziehung zwischen zwei oder mehr Klassen
- An den Enden von Assoziationen sind häufig Multiplizitäten vermerkt. Diese drücken aus, wie viele dieser Objekte in Relation zu den anderen Objekten dieser Assoziation stehen.
- Beispiel:
  - Kunde hat (ein oder mehrere) Konto (n)

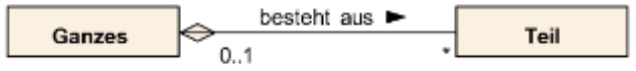


Quelle

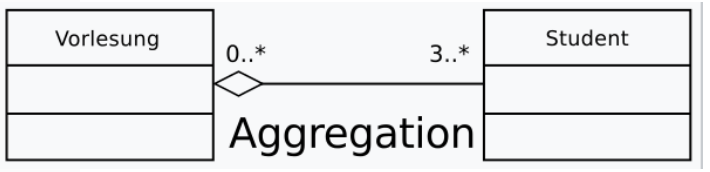
Klassendiagramm - Aggregation

- Beziehung zwischen einem Ganzen und seinen Teilen
  - „Besteht-aus“-Beziehung
- Klasse kann unabhängig von einer anderen Klasse existieren
- An den Enden von Assoziationen sind Multiplizitäten vermerkt
- Das „Ganze“ hat Multiplizität von 0..*

- Beispiel:
  - Eine Vorlesung „besteht aus“ Studenten
  - Studenten können auch ohne Vorlesungen existieren



Quelle

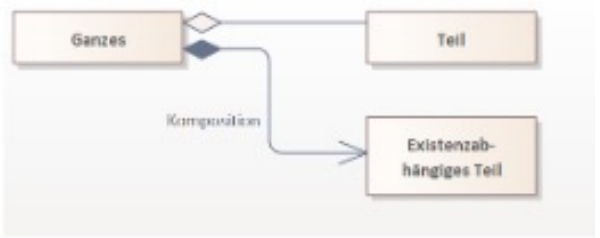


Quelle

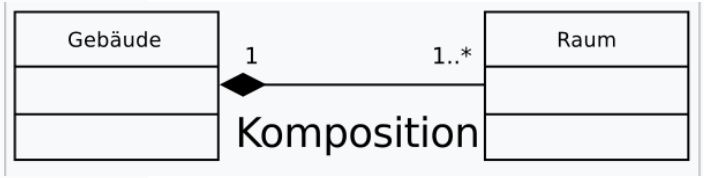
Klassendiagramm - Komposition

- Spezialfall der Aggregation
- Teile hängen von der Existenz des Ganzen ab
- An den Enden von Assoziationen sind Multiplizitäten vermerkt
- Das „Ganze“ hat Multiplizität von 1

- Beispiel:
  - Ein Gebäude „besteht aus“ Räumen
  - Ohne dem Gebäude kann der Raum nicht existieren



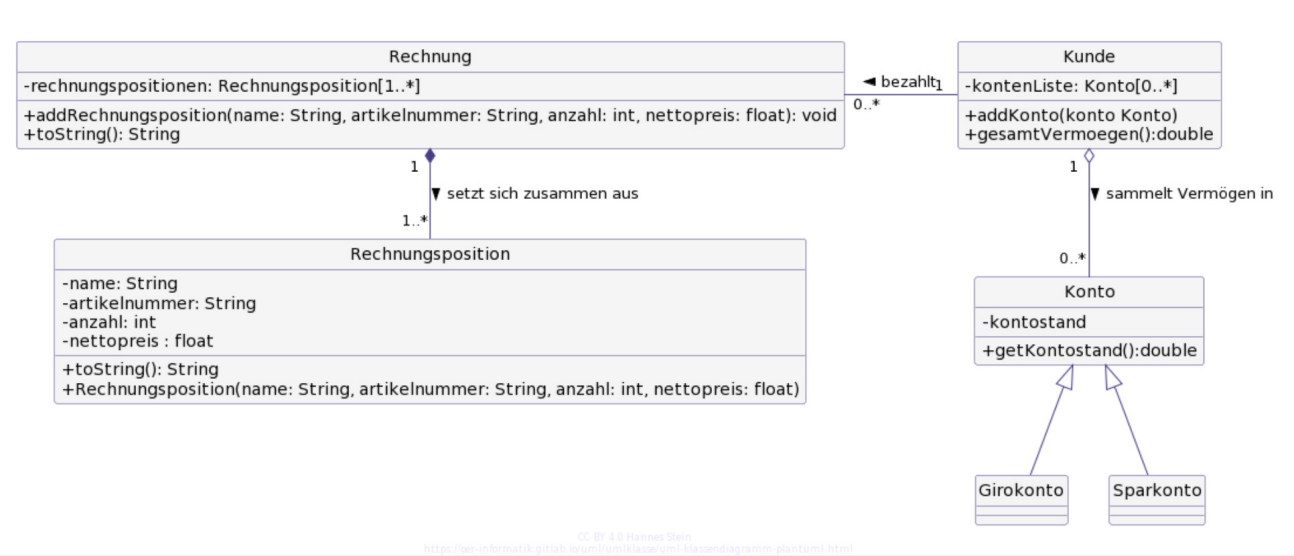
Quelle



Quelle

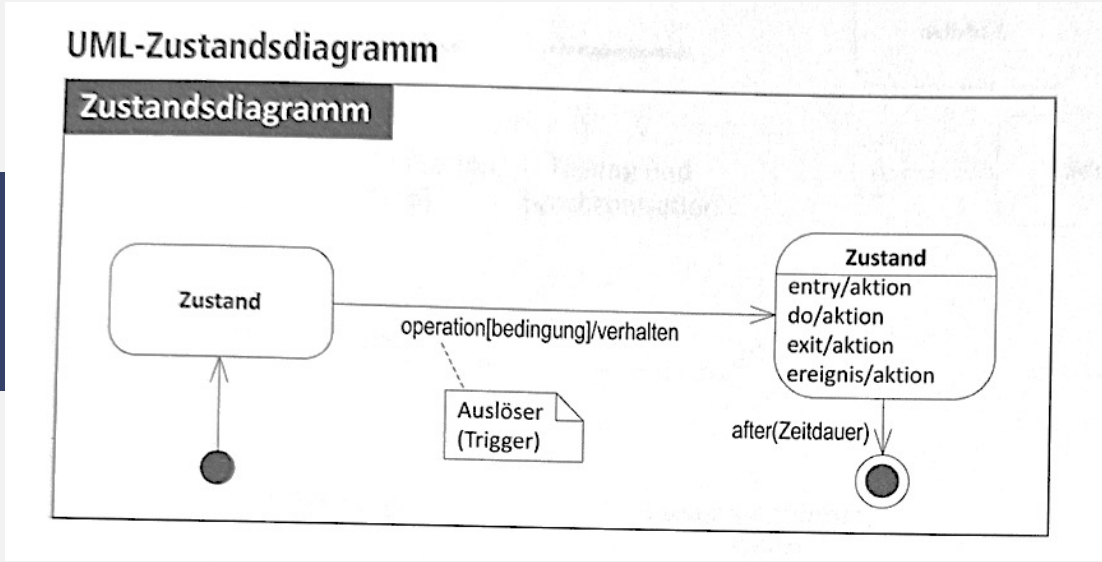
Klassen-diagramm

Einfaches Beispiel

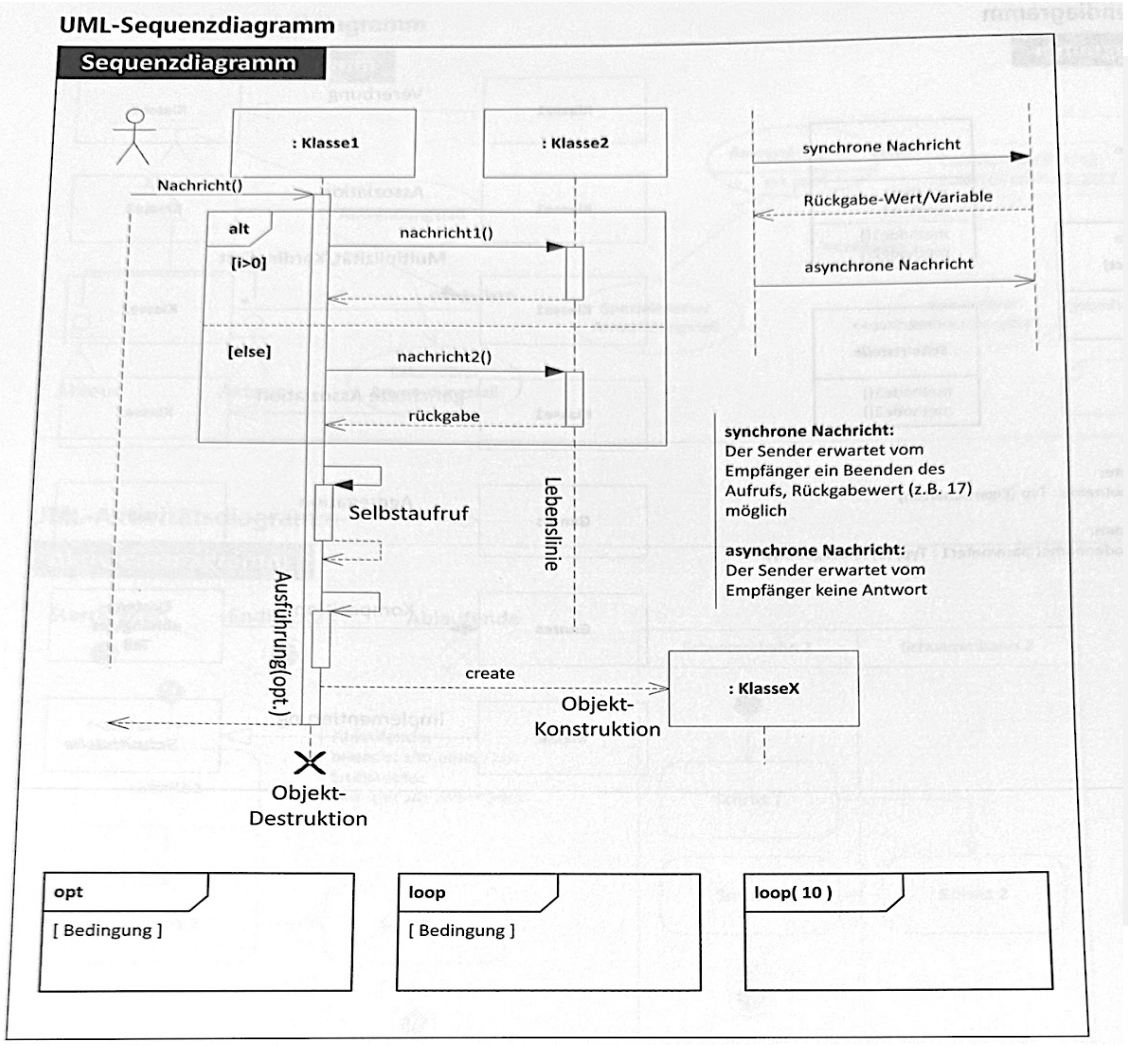


Quelle

Zustands-diagramm



Sequenz-diagramm



Übersicht

Hardware

- CPU
- BUS
- Speicher
- Adressierung

CPU (Central Processing Unit)

- Führt Instruktionen aus dem Speicher
- Steuert den Datenfluss zwischen Speicher, Ein-/Ausgabe und anderen Komponenten

- **Wichtige Aspekte für Entwicklung:**

Taktfrequenz (GHz)	**Kernanzahl**:	**Cache**:	**Instruction Set Architecture (ISA)**:
• beeinflusst, wie viele Instruktionen pro Sekunde abgearbeitet werden können	• Mehr Kerne → parallele Verarbeitung (relevant für Multithreading, Parallel Programming)	• Sehr schneller Zwischenspeicher, minimiert Zugriffszeiten auf RAM	• z. B. x86, ARM – bestimmt maschinennahe Programmierung und Compiler-Optimierung

BUS

- Datenübertragungssystem, das CPU, Speicher und Peripherie verbindet
- **Typen:**
  - **Datenbus** → Transportiert Daten zwischen Komponenten
  - **Adressbus** → Überträgt Speicheradressen
  - **Steuerbus** → Überträgt Steuersignale (z. B. Lese-/Schreibbefehl)
- **Wichtige Aspekte für Entwicklung:**

Busbreite	Bandbreite & Latenz	Bottlenecks
• z. B. 32-Bit, 64-Bit • bestimmt maximal adressierbaren Speicherbereich	• beeinflussen Performance bei speicherintensiven Anwendungen	• Viele Datenoperationen → Bus kann Engpass werden

Speicher

- **Hierarchie:**

Register	Cache	Hauptspeicher (RAM)	Sekundärspeicher
• direkt in der CPU, extrem schnell, klein	• schnell, aber begrenzt	• mittlere Geschwindigkeit, mehrere GB	• SSD/HDD – langsam, aber viel Speicherplatz

- **Speicherarten:**

ROM	RAM
• Fest gespeicherte Daten	• Flüchtiger Arbeitsspeicher

- **Wichtige Aspekte für Entwicklung:**

Speicherzugriffszeiten:	Speicherverwaltung:	Speicherlecks
• Cache-Zugriffe sind viel schneller als RAM-Zugriffe	• Statische vs. dynamische Speicherzuweisung (Heap/Stack)	• vermeiden (C/C++ kritisch, Garbage Collector in Java/Python entlastet)

Adressierung

- Art und Weise, wie die CPU Speicherstellen anspricht
- **Arten:**

Physische Adressierung	Virtuelle Adressierung	Lineare Adressierung	Segmentierte Adressierung
• Tatsächliche Hardware-Adresse	• Betriebssystem abstrahieret Adressen • Memory Management Unit – MMU	• Einfacher Adressraum	• Unterteilung in Segmente • ältere Architekturen

- **Für Entwickler relevant:**

64-Bit-Systeme	**Pointer** (in C/C++)	Speicherlayout
• können mehr Speicher adressieren als 32-Bit-Systeme	• sind direkte Speicheradressen – wichtig für Performance, aber fehleranfällig	• beeinflusst Datenstrukturen (z. B. Alignment, Padding)