LEHRSTUHL 14
SOFTWARE ENGINEERING

Vorlesung (WS 2013/14)

Softwarekonstruktion

Prof. Dr. Jan Jürjens

TU Dortmund, Fakultät Informatik, Lehrstuhl XIV

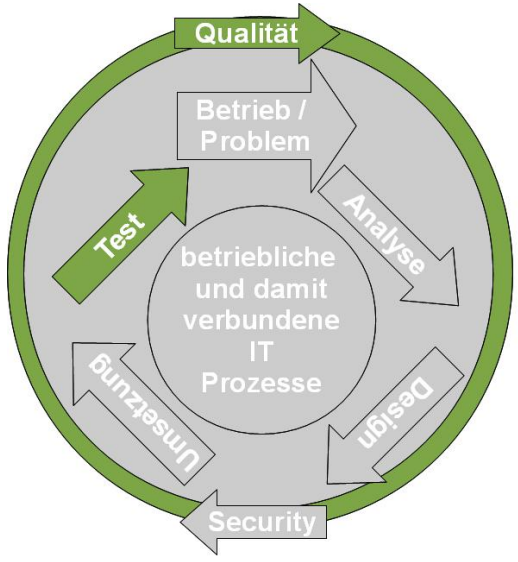
Teil 1.0: Modellbasierte Softwareentwicklung: Einführung

v. 06.11.2013

1.0 Modellbasierte Softwareentwicklung: Einführung

LEHRSTUHL 14
SOFTWARE ENGINEERING

- Modellgetriebene SW-Entwicklung
  - Einführung
  - Object Constraint Language (OCL)
  - Modellbasierte Softwareentwicklung
  - Eclipse Modeling Foundation (EMF)
- Qualitätsmanagement
- Testen
- Modellbasierte Sicherheit



Inkl. Beiträge von Prof. Volker Gruhn (Universität Duisburg-Essen).

Literatur:

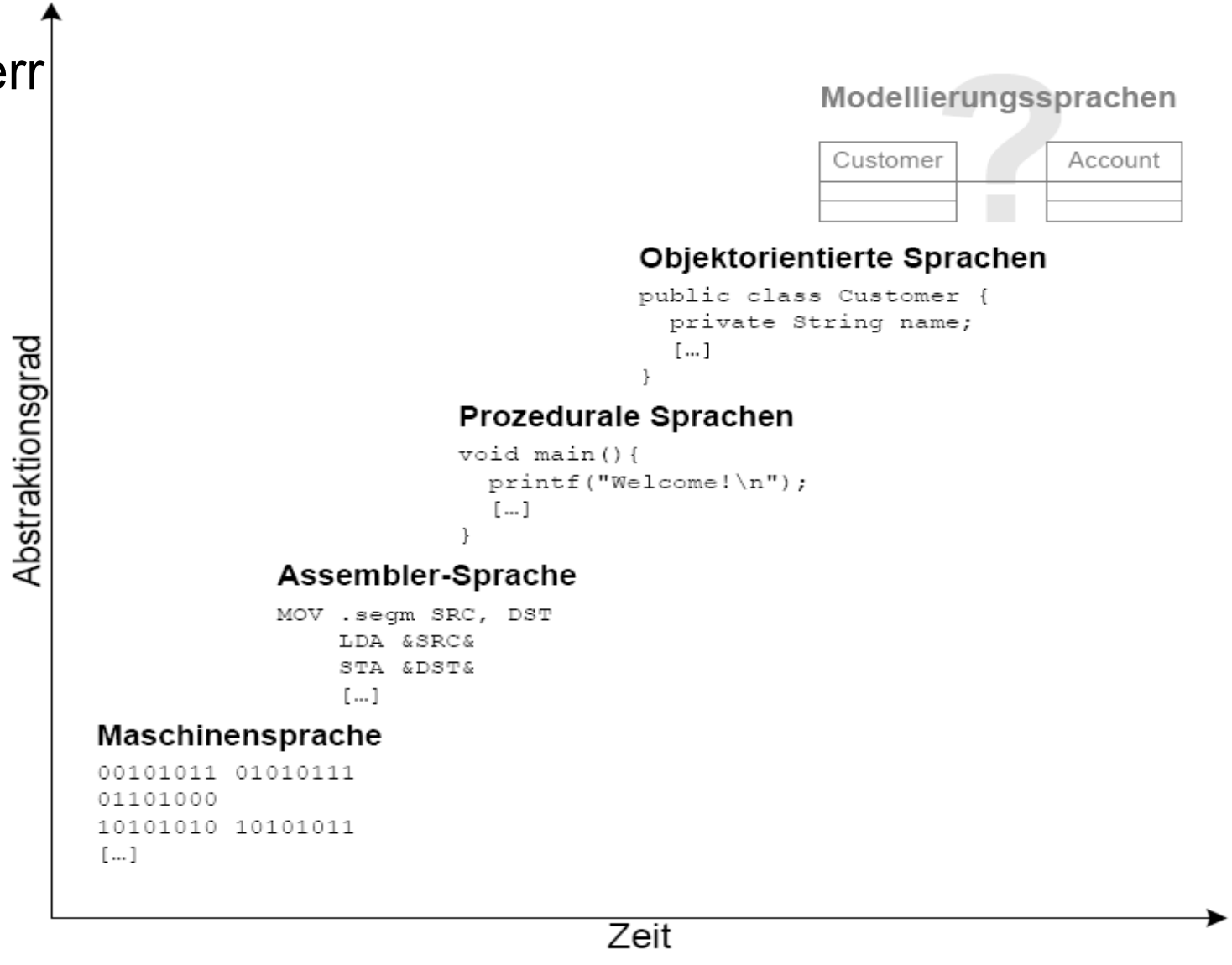
V. Gruhn: **MDA - Effektives Software-Engineering**. (s. Vorlesungswebseite)

- Kapitel 1-2

Geschichte der Softwareentwicklung

LEHRSTUHL 14
SOFTWARE ENGINEERING

- Komplexitäts-beherrschung als **das** Problem.
- Abstraktion als fortwährender Trend.
- Zunehmende Gewichtung von Modellen.



Software-Komplexität: Beispiel Automobil

Beispiele für **Zunahme der Komplexität** von Software:

- **Eingebettete Software** im Kfz:
  - 1970: ca. 100 LOC (Lines of Code)
  - Heute: 1.000.000 LOC, Premiumfahrzeuge 10.000.000 LOC.
- Anstieg der Anzahl von **Electronic Control Units** (ECU) von der letzten zur neuen Mercedes S-Klasse: 64% von 45 ECUs auf 72 ECUs.
- Länge und Gewicht des VW Phaeton Kabelbaums: 3960m, 64kg.
- Beim Ausfall der Bremslichter beim US-New Beatle **blockiert gesamte Automatik.**

Problem Komplexität

Zunehmende Komplexität

Technische Komplexität
- Umfang der Datenmodelle
- Verteilte Implementierung
- Heterogenität der Infrastruktur (Kommunikationsmedien, Protokolle Betriebssysteme / Plattformen)

Funktionale Komplexität
- Umfang der Funktionalität
- Diversifizierung der Funktionalität
- Mensch-Maschine Schnittstelle

Entwicklungskomplexität
- Einflussnahme des Kunden
- Entwicklung in Zulieferketten (Integrationsaufwand)
- Qualitätsanforderungen
- Innovationsdruck

Qualität, Kosten, Termine

Qualität
- Nutzersicht: Funktionsvielfalt, Usability, Sicherheit, Performanz
- Entwicklungssicht: Wartbarkeit,
- Wiederverwendbarkeit

Kosten
- Entwicklungskosten
- Vermarktbarkeit
- Kosten im Gesamtlebenszyklus (Entwicklung, Inbetriebnahme, Wartung) Total Life Cycle Costs

Entwicklungszeit
- Time to Market
- Reaktionszeit bei Änderungen

Komplexität beherrschen

Ansätze, um **Komplexität zu beherrschen:**

- **Abstraktion:**
  - Ausblenden von Detailinformation.
  - Einsatz von geeigneten SW-Modellen.
- **Strukturierung / Modularisierung:**
  - Gliederung und Aufteilung in klar abgegrenzte Unterstrukturen / Module.
  - Partitionierung der Aufgaben (Dekomposition).
  - Einsatz von geeigneten SW-Modellen.
- **Methodik und Systematik:**
  - Verwendung bewährter Verfahren und Lösungsmuster.
  - Systematisierung des Entwicklungsprozesses.
  - Code Ebene: Design Pattern, Architektur: Referenzarchitekturen.

Weitere akute Probleme: Steigende Anforderungen

Steigende Anforderungen:

- Anforderungen an **Leistungsfähigkeit**, **Zuverlässigkeit** und **Qualität**.
- **Kurze Technologiezyklen**, für Hardware-/Software-Plattformen.
- Häufige **Anforderungsänderungen**.
- Hoher Druck zur **Kostenreduzierung**.
  - insbesondere in Zeiten konjunktureller Schwächeperioden

Weitere akute Probleme: Fachlichkeit vs. Technik

Dominierung von Fachlichkeit durch Technik:

- Umsetzung fachlicher Basiskonzepte dominiert durch Technik.
- Anwendungsentwickler:
  - Benötigen **umfangreiches technisches Wissen...**
  - ...statt sich auf fachliche Anwendungsdomäne zu konzentrieren.
- Fachabteilung und Entwickler:
  - **Verständigung** auf völlig unterschiedlichen Abstraktionsebenen.
  - Abstraktionsniveau derzeitiger Entwicklungsansätze zu niedrig.
- **Durch Fachlichkeit bestimmte Entwicklung** in weiter Ferne.

Weitere akute Probleme: Fehlende Durchgängigkeit

LEHRSTUHL 14
SOFTWARE ENGINEERING

Methodischer Bruch zwischen Analyse, Design und Implementierung:

- Viele Modelle nach Beginn Implementierungsphase **nicht aktualisiert.**
  - Macht Modelle nahezu nutzlos.
- **Erschwert Einarbeitung** neuer Mitarbeiter in späten Phasen.
- **Erhöht Rüstzeiten** in Betriebs- bzw. Wartungsphase um ein Vielfaches.

Fehlende Traceability:

- Fehlende Durchgängigkeit für Lebenszyklus.
- Requirements Engineering: In Bezug auf Anforderungen, fehlende
  - **Nachvollziehbarkeit**
  - **Rückverfolgbarkeit**

Akute Probleme: Folgerung

- Bedarf nach **höheren Abstraktionsniveau**.
- Potenzial, Gleichförmigkeiten in verdichteten Form zusammenzufassen.
- Forderung nach **durchgängigeren Auswahl der Ausdrucksmittel.**
- **Modelle:**
  - Abstrahieren und fokussieren auf Wesentliche.
  - Schlagen Brücke von fachlicher Problemwelt in technische Lösungswelt.

Kernideen:

- Modelle: **Zentrales Artefakt** im SW-Prozess.
  - **Konsequente Nutzung** von erster Phase des Lebenszyklus zur letzten (Anforderung bis Wartung).
- **Vermeidung von Modellbrüchen** im SW-Prozess.
- **Einsatz von Softwaremodellen** mit fachlicher Semantik:
  - Fachliche Anforderung von konkreter Technologie entkoppeln.
  - Wiederverwendung fachlicher Aspekte.

Häufig Verwendung der **Unified Modeling Language (UML)** für Modellierung (aber nicht notwendig).

Diskussion: Modellbasierte Szenarien

LEHRSTUHL 14
SOFTWARE ENGINEERING

Für welche Zwecke könnte man Ihrer Meinung nach Modelle (z.B. in UML) in der modellbasierten Softwareentwicklung verwenden ?

Modellbasierte Szenarien

Dokumentation und Kommunikation mit Modellen:

- **Analysemodelle**: Fachexperte und SW-Architekt.
- **Entwurfsmodelle**: SW-Architekt und SW-Entwickler.
- **Implementierungsmodelle**: Dokumentation von Systemen.

Spezifizieren mit Modellen:

- **Verfeinerung** und **Spezialisierung** des Modells in Entwurfsphase.
- Verbindliche Spezifikation zwischen **Auftraggeber** und IT-Firma.
- Vorlage zur Implementierung durch **SW-Entwickler**.
- Generieren von **Dokumenten** aus Modell.

Testen mit Modellen:

- Ableiten von **Integrations-** und **Abnahmetests** aus fachlichem Modell.
- **Automatische Generierung** von technischen Testfällen (Junit).

Modellbasierte Szenarien

Simulieren mit Modellen:

- **Simulation** von Dynamik zur Validierung von Systemteilen.
- **Frühes Feedback** → Auswirkung von Designentscheidungen.
- Systemverhalten: z.B. **Nebenläufige Prozesse.**

Programmieren mit Modellen:

- **Modellieren** statt Programmieren: Teile des Programmcodes generieren.
- **Generatoren** für konkrete technische Bereiche (z.B. DB-Schema aus XML).
- Generatoren für komplette **Schichten** eines Systems (Persistenzschicht samt Konfigurationen und Datenobjekte).
- Generatoren für spezielle aber **schichtenübergreifende** Bereiche eines Systems (z.B. UI, Validierung und Persistenz aus XML-Datei).
- Generatoren für speziellen **Projektzweck** (z.B. Kapselung kundenspezifischer Besonderheiten, Produktlinien).

Modellbegriff und SW-Modelle

Modell:

- Abbildung: Welt → Diskrete Struktur.
- „A model is a simplification of reality“ [BRJ01].
- „A *model* is a *simplification of a system* built with an intended goal in mind. The model should be able to answer questions in place of the actual system.“ [Beziv01].

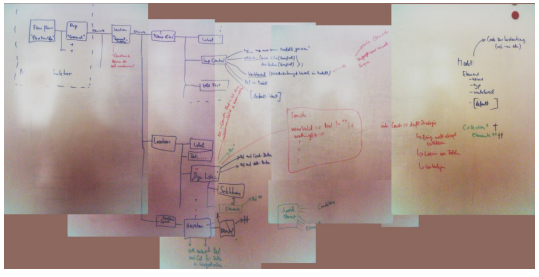
Einige Aspekte, die modelliert werden:

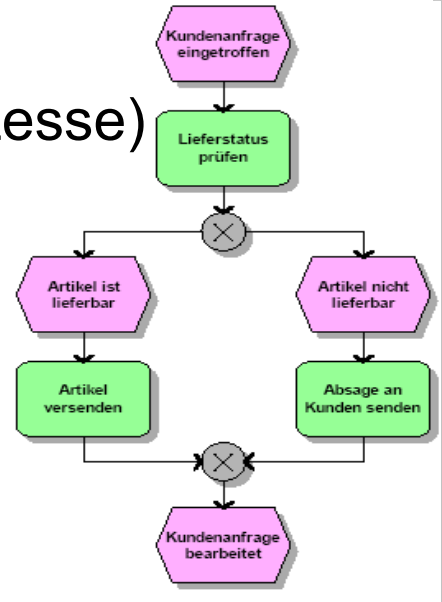
- Struktur.
- Beziehungen.
- Verhalten.

Modelle im Software Engineering: Beispiele

Grafische Modelle (Schwerpunkt auf menschlichen Nutzer)

- Ereignisgesteuerte Prozessketten (EPK) (für Geschäftsprozesse)
- Petrinetze
- Unified Modelling Language
- Informelle Skizzen





Modelle auf **Textbasis**:

- XML-Schema (Schwerpunkt auf maschineller Verarbeitung)
- Matlab Code (Kompromiss zwischen menschlicher und maschineller Nutzung)

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XM
     <xs:element name="auftrag">
        <xs:complexType>
           <xs:all>
              <xs:element name="kundennummer"
              <xs:element name="auftragsdatum
              <xs:element name="ausführungsda
              <xs:element name="auftragsposit
                 <xs:complexType><xs:sequence
ref="auftragspositiontype"/></xs:sequenc
              </xs:element>
              <xs:element name="kundenanschri
                 <xs:complexType><xs:sequence
              </xs:element>
              <xs:element name="rechnungsansc
                 <xs:complexType><xs:sequence
              </xs:element>
              <xs:element name="installations
```

Grenzen zwischen **Modell** und **Code** sind fließend;
kann Code als Modell auffassen.

Könnte evt. auch Textdokumentation als Modell auffassen;
bislang aber nicht vollständig verarbeitbar
=> kein Modell im Sinne modellbasierte Entwicklung

Syntax vs. Semantik von Modellen

Syntax: Sichtbare Modellelemente (konkrete Syntax) oder deren abstrakte Repräsentation (abstrakte Syntax)

Semantik: Bedeutung, die durch das Modell ausgedrückt werden soll, abhängig von Art des Modells, z.B.:

- UML-Klassendiagramm: Beziehungen zwischen Klassen
- UML-Statechart: Verhalten eines Systemteiles

Für Modell-Verarbeitung in modellbasierter Entwicklung benötigt Werkzeug relevante Information über Semantik, z.B.:

- Testgenerierung aus UML-Statechart: intendiertes Verhalten

Syntax: werkzeugrelevante Standard-Formate wie XMI (XML-Dialekt für Speichern der Syntax von UML-Modellen).

Semantik: keine einheitlichen Formate.

- UML: Object Constraint Language (OCL).
- Alternativen: Petrinetze, Abstract State Machines, Logik erster Stufe, Temporale Logik, ...

Wiederholung: UML

Unified Modeling Language (**UML**, http://omg.org/uml):

- Modellierung, Dokumentation, Spezifizierung und Visualisierung von komplexen Softwaresystemen.
- **Standard** der Object Management Group (OMG).
- **Unterschiedliche Modellierungskonzepte** auf einheitlicher Basis:
  - 14 Diagrammtypen in Version 2.2 (http://omg.org/spec/UML/2.2)
- **Modell** vs. **Diagramm**:
  - UML-Modell besteht aus einem oder mehreren Diagrammen.
  - Diagramm entspricht bestimmter Sicht auf Modell, stellt Teil der im Modell enthaltenen Information dar.
  - Modellelemente können in mehreren Diagrammen vorkommen. (=> Konsistenz oft nicht-trivial).

Zusammenfassung und Ausblick

Dieser Abschnitt: Einführung und Überblick in „Modellbasierte Software-Entwicklung“. Wichtige Punkte:

- Aktuelle Herausforderungen
- Modellbasierte Softwareentwicklung
- Modellbegriff und SW-Modelle
- Semantik von Modellen

Nächster Abschnitt: Object Constraint Language (OCL) zur Unterstützung der UML-Modellierung.

Anhang
(Weitere Informationen und Beispiele)

Geschichte der Softwareentwicklung
Anfänge

- **Maschinensprachen:** 50er-Jahre
  - Computer-Welt durch teure Hardware-Ressourcen bestimmt.
  - Software hatte beherrschbare Größe; in Maschinensprache binär kodiert.
- **Assembler-Sprachen:**
  - Komplexere Software-Systeme nicht dazu geeignet, um in Binärcode entwickelt zu werden.
  - Löste Maschinensprachen ab.
  - Mnemonische Symbole, Marken und Makros als Arbeitserleichterung.

Höhere Programmiersprachen:

- Zunehmend Programmiersprachen gefragt.
- **FORTRAN** (FORmula TRANslator):
  - 1954-1958 von Jim Backus entwickelt.
  - **Erste höhere Programmiersprache**.
  - Für **mathematisch-technische Probleme** konzipiert.
  - Vorläuferin vieler weiterer höherer Programmiersprachen.

Software-Krise und Software-Engineering:

- Hardware-Preise fielen.
- **Bedarf an Software wuchs.**
- Programmierte Systeme größenmäßig **komplex und unhandhabbar.**
- NATO-Konferenz (1968): **Software-Krise** wurde ausgerufen.
  - Forderung nach angemessenen Ingenieursdisziplin.
  - In dieser Zeit Begriff des Software-Engineering geprägt.

- **Strukturierte Programmierung:**
  - Durch Kontrollstruktur- und Fallunterscheidungsmöglichkeiten.
    - z.B. etwa in Pascal oder in C.
- **Systematische Methoden:**
  - Strukturierte Analyse (SA)[1].
  - Strukturierte Design (SD)[2].

1Tom DeMarco: Structured Analysis and System Specification. Yourdon Press, 1978.
2 Edward Yourdon und Larry L. Constantine: Structured Design – Fundamentals of a Discipline of Computer Program and Systems Design. Prentice Hall, 1979.

Geschichte der Softwareentwicklung
SA & SD

- **Strukturierte Analyse:**
  - Hierarchisch angeordnete Datenflussdiagramme zur abstrakten Modellierung von Prozessen.
  - Mini-Spezifikationen zur Beschreibung von Prozessen.
  - Data Dictionaries für einheitliche Datendefinitionen.
- Funktionen in strukturiertem Design in hierarchisch aufgebaute Module zerlegt. → In **Strukturdiagrammen** festgehalten.
- Diagramme zur
  - **Visualisierung von Programmabläufen** und
  - **Beschreibung von Schnittstellen**

  zwischen Modulen.
- **Diagramme/Modelle:** Hoher Stellenwert in strukturierten Methoden.
- Markt für **Modellierungs-Werkzeuge:**
  - Computer Aided Software Engineering (CASE).

Wiederverwendungskrise:

- Ende der 80er-Jahre.
- Drastisch gewachsene **Komplexität** in Software-Systemen.
  - Erzwang Umdenken bei der Software-Erstellung.
  - Wunsch nach Wiederverwendbarkeit von Software.
- Gewaltige **Software-Bestände** angehäuft.
  - Problem, diese auch nur teilweise wieder zu verwenden.

Geschichte der Softwareentwicklung
Objektorientiertes Paradigma

- **Objektorientierte Paradigma:**
  - Als **Lösung** der Wiederverwendungskrise.
  - System besteht aus Objekten. Jedes Objekt besitzt definiertes Verhalten, inneren Zustand und eindeutige Identität.
  - Ansatz in 70er-Jahren in wissenschaftlichen Veröffentlichungen erschienen.
  - **SIMULA** (SIMUlation LAnguage):
    - Ole-Johan Dahl und Kristen Nygaard in 60er-Jahren.
    - Ursprünglich für diskrete Ereignis-Simulation entwickelt.
  - **Breite Akzeptanz** bei Smalltalk, C++ und Java.
  - Vererbung und Polymorphie für **Wiederverwendbarkeit.**
- **Design und Analyse:**
  - 90er-Jahre viele neue objektorientierte Methoden
    - Intensivierter Gebrauch von Modellen führte zur Unified Modeling Language (UML)

Weitere akute Probleme: Divergenz der Änderungszyklen

Divergenz der Änderungszyklen:

- **Änderungszyklus** der Technologie vs. Änderungen fachlicher Prozesse.
- **Technische und fachliche Belange** der Entwicklung trennen.
  - Besserer **Investitionsschutz** der Entwicklungsarbeit.
  - Verlängerte **Lebensdauer** der Fachkonzepte.
  - Größere **Flexibilität** und bessere Reaktionsfähigkeit bei ändernden Fachanforderungen.

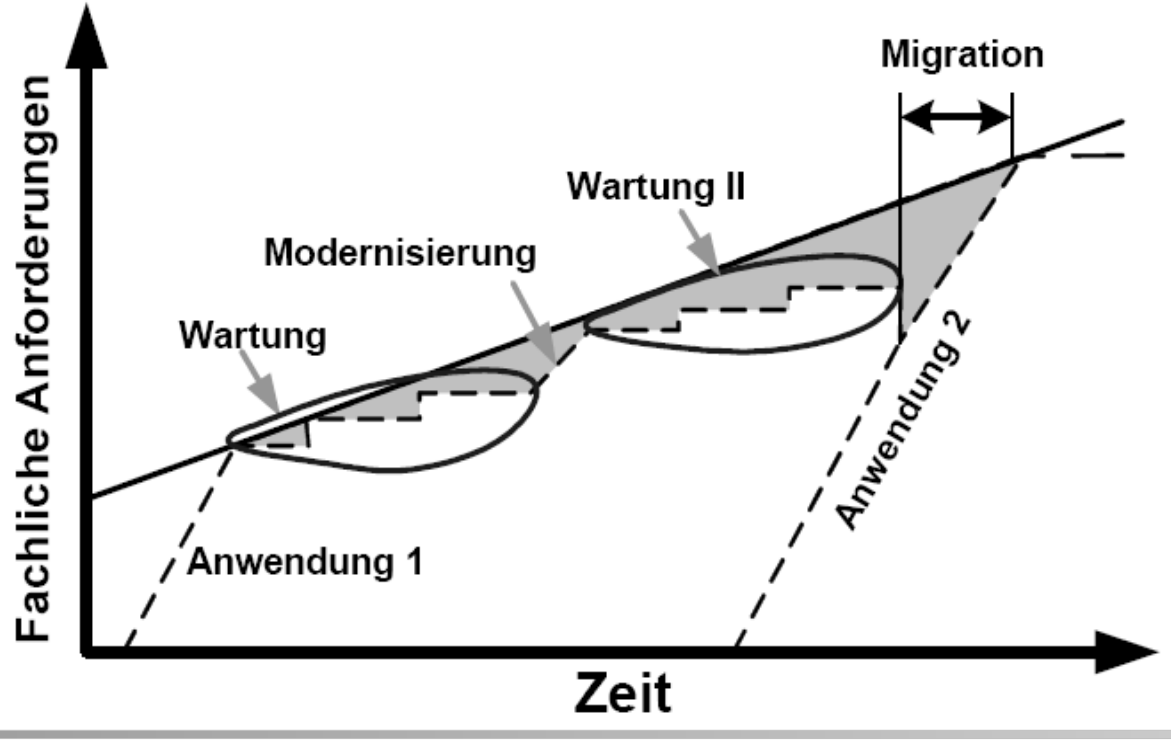
Weitere akute Probleme: Middleware, Legacy

Middleware Babel:

- Praxis: **Heterogene Systemlandschaft** mit vielen verschiedenen Middleware-Technologien.
- **Integrationsprozess** zunehmend schwieriger.

Legacy Crisis:

- Erhöhter Druck zur Modernisierung bereits bestehender Anwendungen.
- Rückstau fachlicher Änderungswünsche.
  - Bis Modernisierung notwendig wird.



SW-Modelle: XML-Schema (XSD)

- **Modellierung** strukturierter Textdaten:
  - **Technisches Modell** von z.B. Schnittstellen, Datenbanken und Konfigurationsdateien.
  - **XML-Schema** definiert Elemente und Attribute von XML-Dateien mit
    - einfachen Datentypen.
    - komplexen Datentypen.
    - Wertebereichen.
    - Reihenfolge und Anzahl von Elementen.
    - XML-Schema ist wiederum XML-Datei.
  - **XML-Dateien:**
    - Strukturierte Beschreibung.
    - Von Menschen und Maschinen verstehbar.

SW-Modelle: XSD Beispiel


```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified" attributeFormDefault="unqualified">
      <xs:element name="auftrag">
         <xs:complexType>
            <xs:all>
               <xs:element name="kundennummer" type="xs:int"/>
               <xs:element name="auftragsdatum" type="xs:date"/>
               <xs:element name="ausführungsdatum" type="xs:date"/>
               <xs:element name="auftragsposition">
                  <xs:complexType><xs:sequence maxOccurs="unbounded"><xs:element
      ref="auftragspositiontype"/></xs:sequence></xs:complexType>
               </xs:element>
               <xs:element name="kundenanschrift">
                  <xs:complexType><xs:sequence><xs:element ref="adressetype"/></xs:sequence></xs:complexType>
               </xs:element>
               <xs:element name="rechnungsanschrift">
                  <xs:complexType><xs:sequence><xs:element ref="adressetype"/></xs:sequence></xs:complexType>
               </xs:element>
               <xs:element name="installationsanschrift">
                  <xs:complexType><xs:sequence maxOccurs="unbounded"><xs:element ref="adressetype"/></xs:sequence></xs:complexType>
               </xs:element>
            </xs:all>
         </xs:complexType>
      </xs:element>
      <xs:element name="auftragspositiontype">
         <xs:complexType>
            <xs:all>
               <xs:element name="beschreibung" type="xs:string"/>
               <xs:element name="unterposition">
                  <xs:complexType><xs:sequence maxOccurs="unbounded"><xs:element ref="unterpositiontype"/></xs:sequence></xs:complexType>
               </xs:element>
            </xs:all>
         </xs:complexType>
      </xs:element>
      <xs:element name="unterpositiontype">
         <xs:complexType><xs:all><xs:element name="beschreibung" type="xs:string"/></xs:all></xs:complexType>
      </xs:element>
      <xs:element name="adressetype">
         <xs:complexType><xs:all>
               <xs:element name="vorname" type="xs:string"/><xs:element name="nachname" type="xs:string"/>
               <xs:element name="strasse" type="xs:string"/><xs:element name="hausnummer" type="xs:int"/>
               <xs:element name="postleitzahl" type="xs:string"/> <xs:element name="stadt" type="xs:string"/>
         </xs:all></xs:complexType>
      </xs:element>
</xs:schema>
```

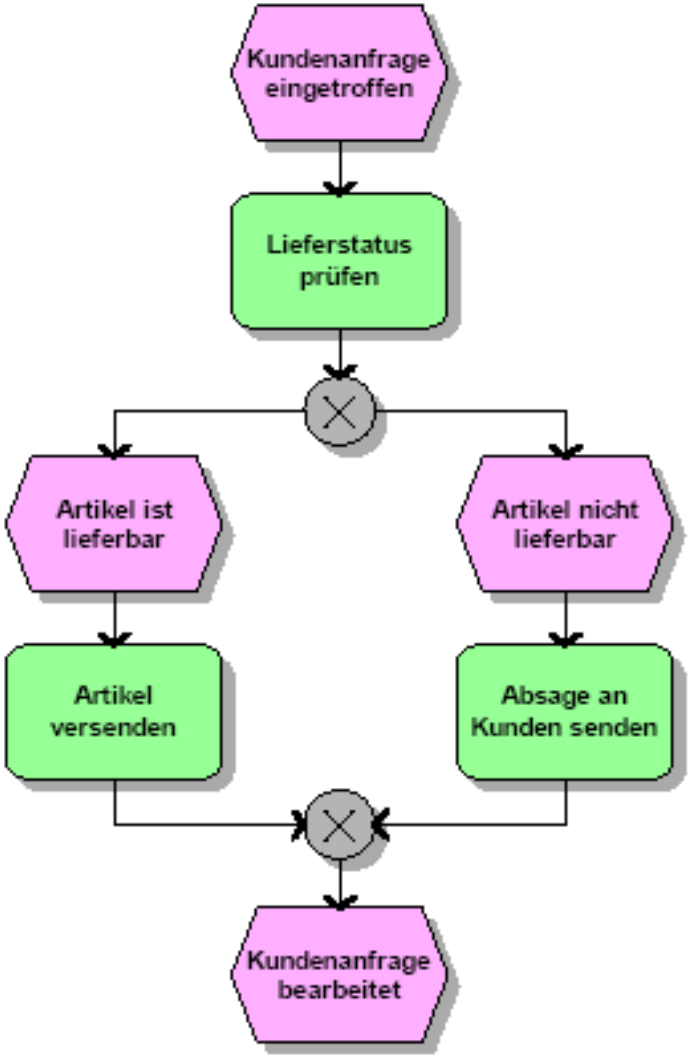
SWK-Modelle: EPK

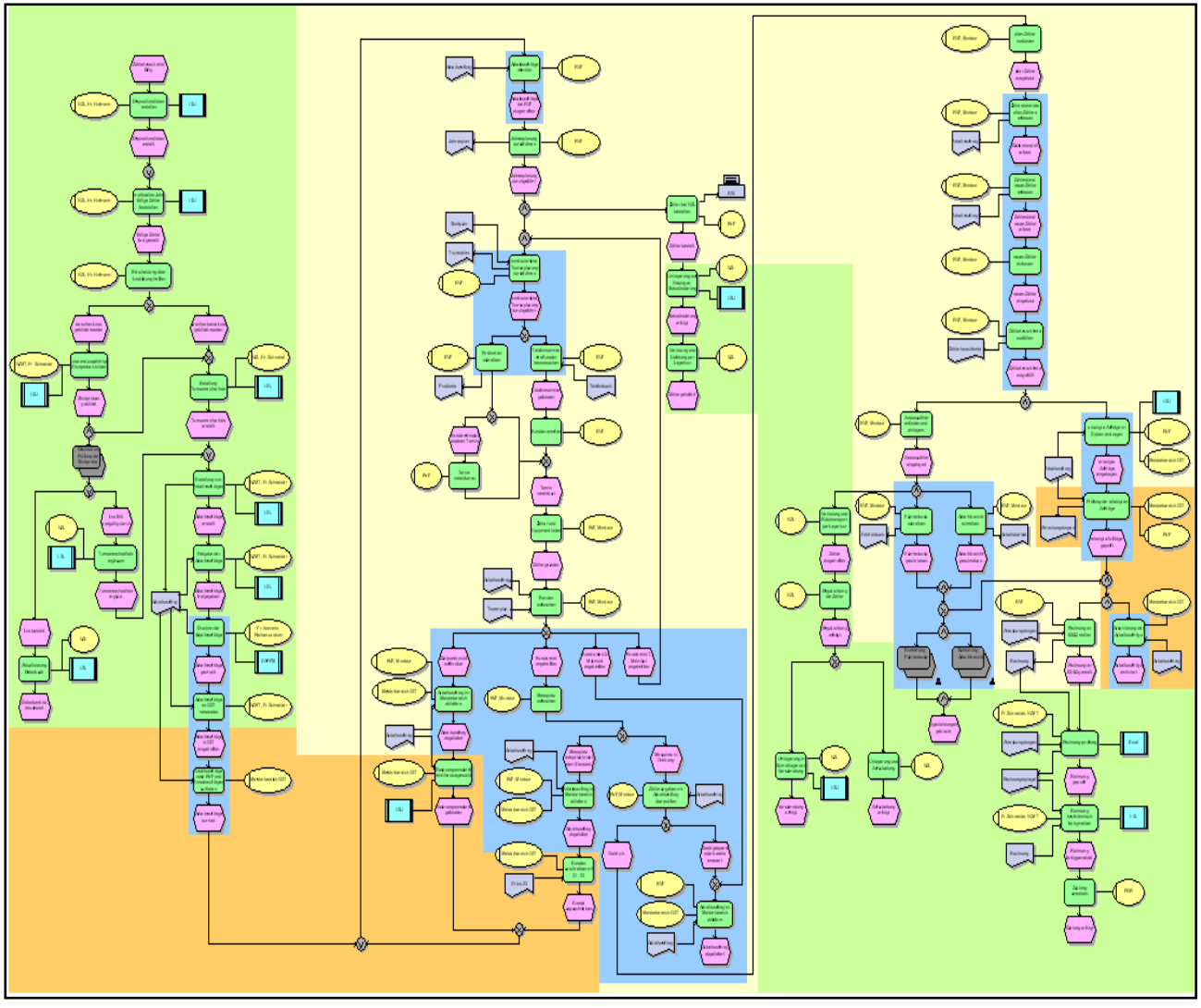
Ereignisgesteuerte Prozessketten:

- Beschreibung von Geschäftsprozessen von Unternehmen.
- Symbole in EPKs:
  - **Ereignis:** Betriebswirtschaftlicher Zustand.
  - **Funktion:** Aktivität eines Geschäftsprozesses.
  - **Verknüpfungsoperatoren:** AND, XOR, OR.
  - **Verbindung** der Elemente per Kontrollfluss-Pfeil.

SWK-Modelle: EPK - Beispiel

LEHRSTUHL 14
SOFTWARE ENGINEERING



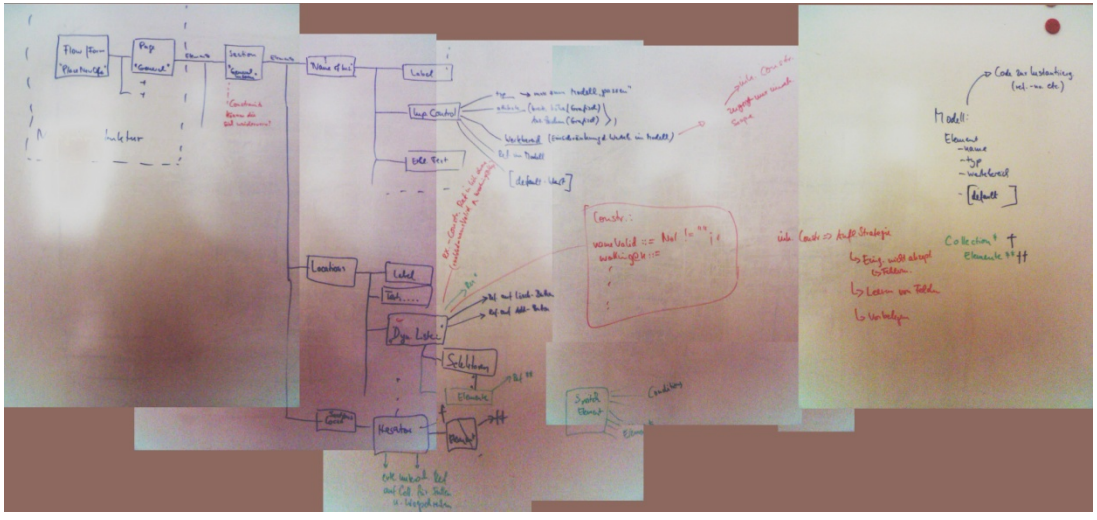


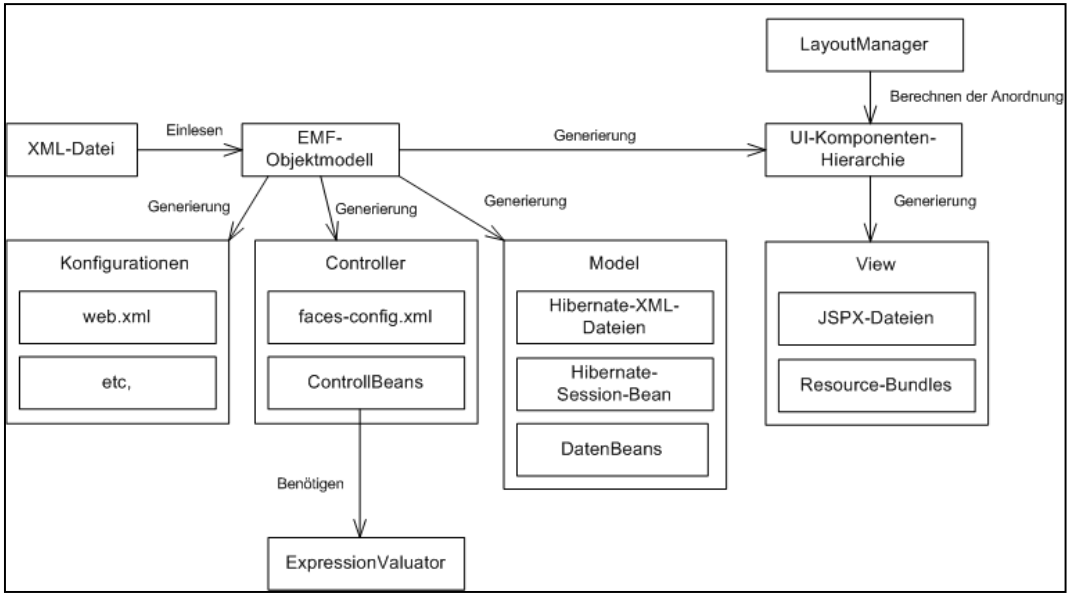
SW-Modelle Skizzen:

Skizzen:

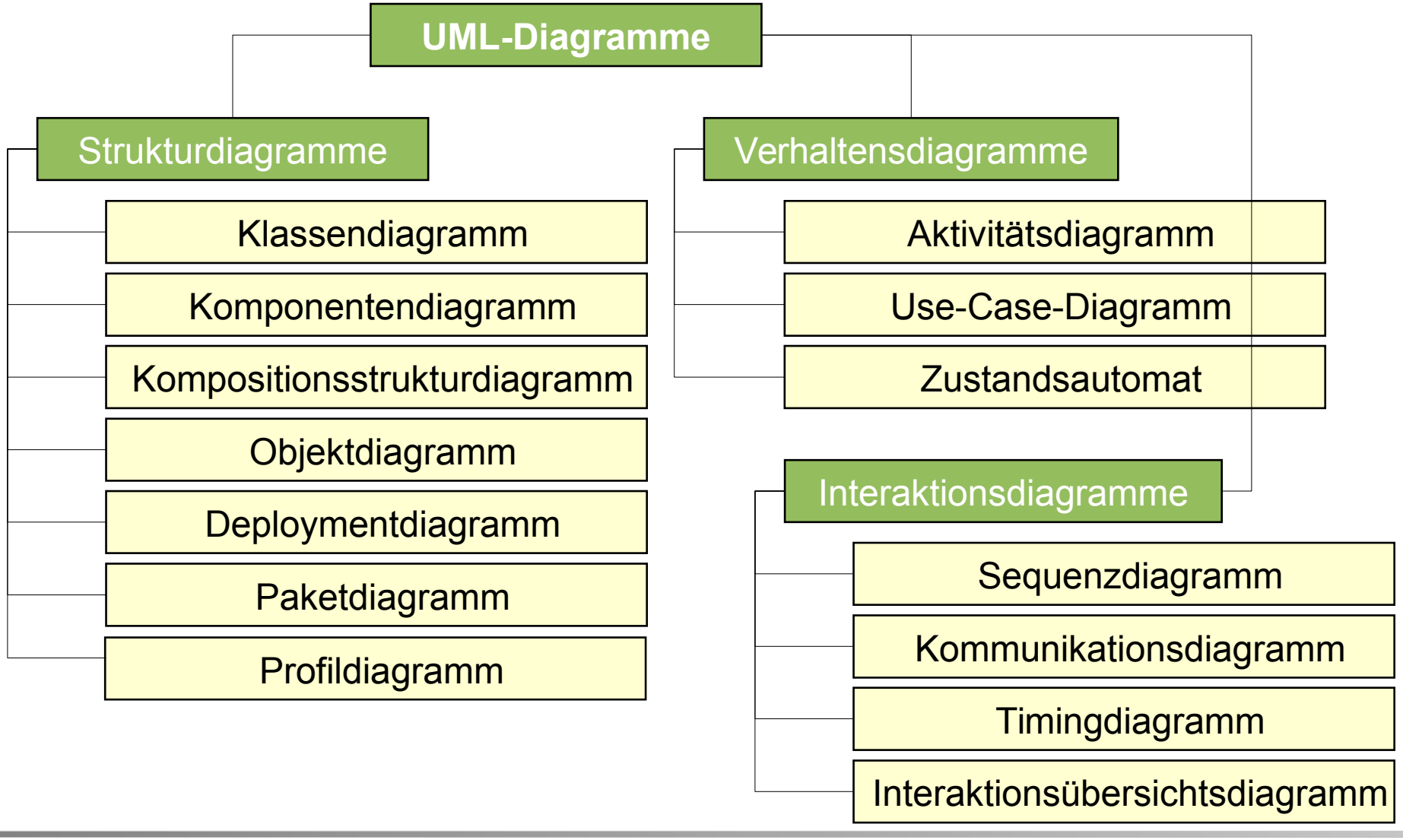
- Per Hand (Papier, Whiteboard) oder mit Tool (Powerpoint, Visio) erstellte **Graphiken.**
  - Verdeutlichen **fachliche oder technische Zusammenhänge.**
    → Ohne klar definierte Semantik.
    - **Interpretation der Skizzen** von nicht an Erstellung beteiligten Personen nicht gewährleistet.
- **Wichtigste Artefakte der Kommunikation** innerhalb Projektteams.

SWK-Modelle: Skizzen - Beispiel





Wiederholung: Arten von UML-Diagrammen



Wiederholung: Einige UML-Diagrammtypen

Anwendungsfalldiagramm: Enthält die Anforderungen an das System

Klassendiagramm: Datenstruktur des Systems

Zustandsdiagramm: dynamisches Verhalten der Komponenten

Aktivitätsdiagramm: Steuerung des Ablaufs zwischen den Komponenten

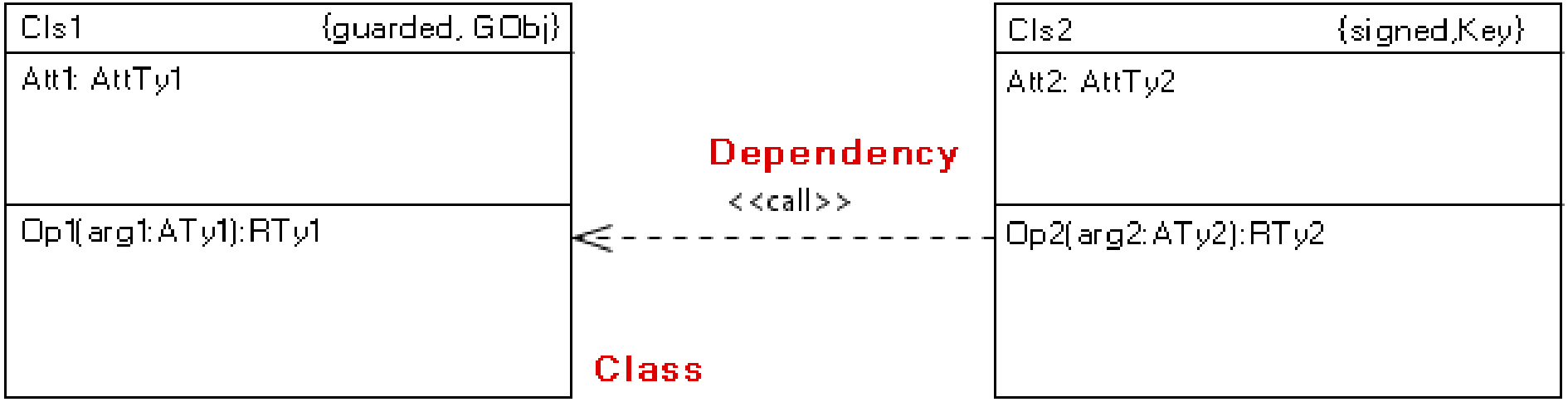
Sequenzdiagramm: Interaktion durch Nachrichtenaustausch

Verteilungsdiagramm: Physikalische Umgebung

Paket/Subsystem: Fasst Diagramme eines Systemteils zusammen

Klassenstruktur des Systems.

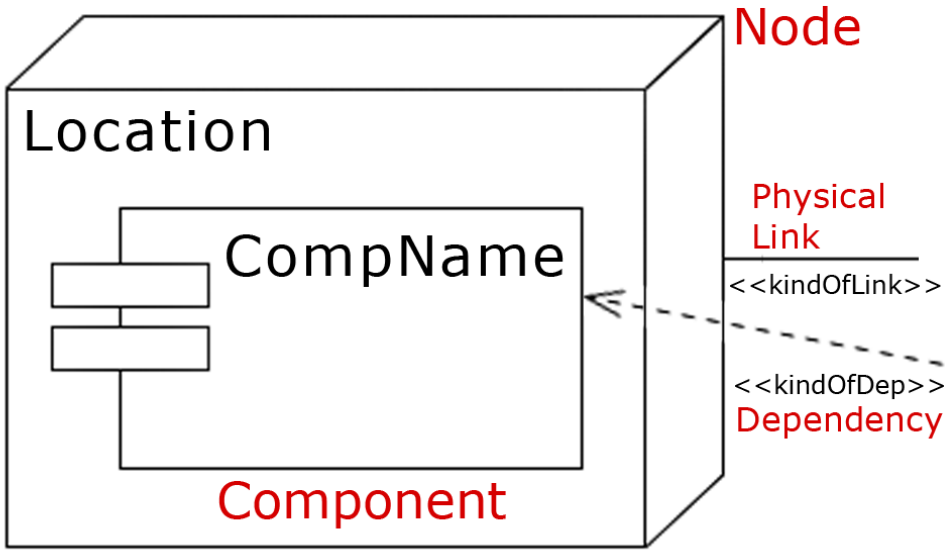
Klasse mit Attributen und Operation / Signalen,
Beziehungen zwischen Klassen.



Wiederholung UML:
Verteilungsdiagramm

- Problem:
  - Es soll die Verteilung der Software des Systems auf die Hardware beschrieben werden.
- Diese zentrale Frage beantwortet das Diagramm:
  - Wie sieht das Einsatzumfeld (Hardware, Server, Datenbanken etc.) des Systems aus?
  - Wie werden die Komponenten zur Laufzeit wohin verteilt?

Wiederholung UML:
Verteilungsdiagramm

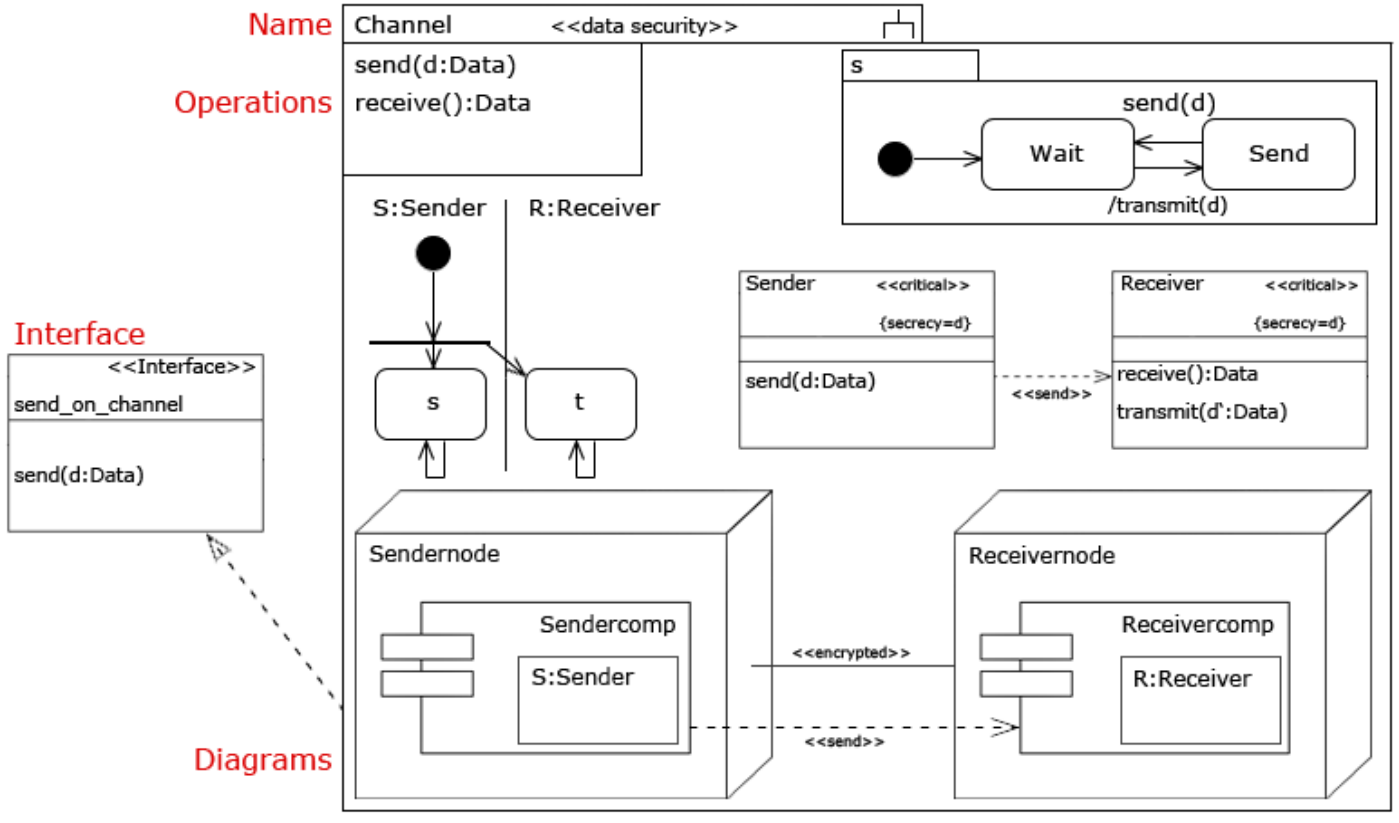


Erklärt die physikalische Ebene, auf der das System implementiert wird.

Wiederholung UML:
Paketdiagramm

- Problem:
  - Große Softwaresysteme mit mehreren 100 Klassen lassen sich nicht mehr von einer Person überblicken.
- Diese zentrale Frage beantwortet das Diagramm:
  - Wie kann ich mein Modell so schneiden, dass ich den Überblick bewahre?
- Diese Stärken hat das Diagramm:
  - organisiert das Systemmodell in größeren Einheiten durch logische Zusammenfassung von Modellelementen
  - Modellierung von Abhängigkeiten und Inklusionen ist möglich

Wiederholung UML:
Paketdiagramm



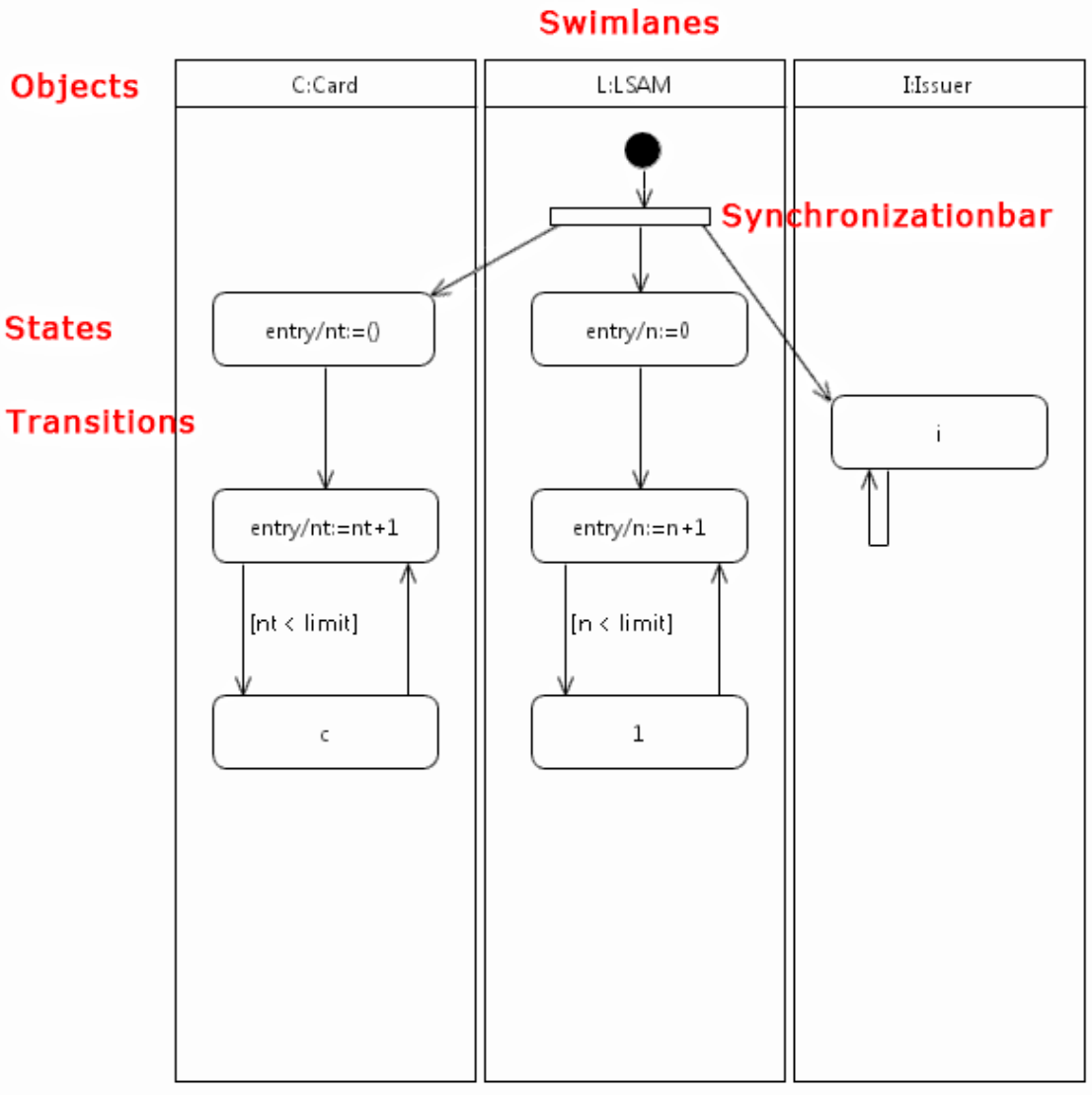
Kann benutzt werden, um UML Element zu einer Gruppe zusammen zu fügen.

Wiederholung UML:
Aktivitätsdiagramm

- Problem:
  - Es sollen Abläufe, z.B. Geschäftsprozesse, modelliert werden. Im Vordergrund steht dabei eine Aufgabe, die in Einzelschritte zerlegt werden soll.
  - Es sollen Details eines Anwendungsfalles festgelegt werden.
- Diese zentrale Frage beantwortet das Diagramm:
  - Wie realisiert mein System ein bestimmtes Verhalten?

Wiederholung UML: Aktivitätsdiagramm

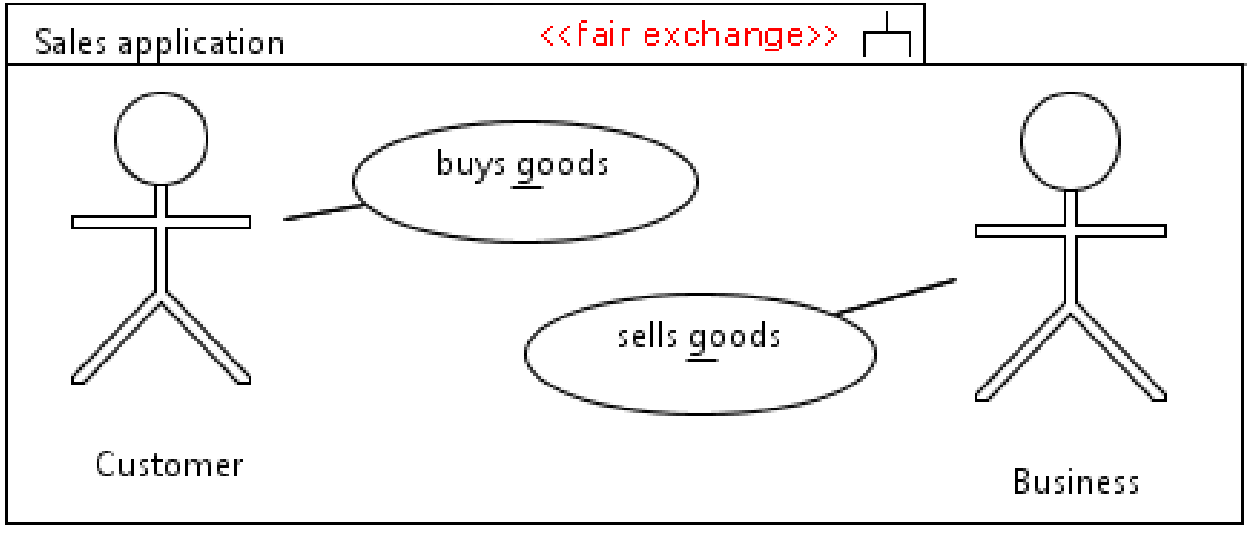
Spezifiziert den Kontrollfluss zwischen Komponenten desselben Systems. Ist auf einer höheren Abstraktionsebene als Zustands- und Sequenzdiagramme.



Wiederholung UML:
Anwendungsfalldiagramm

- Problem:
  - Das externe Verhalten des Systems soll aus Nutzersicht dargestellt werden.
- Diese zentrale Frage beantwortet das Diagramm:
  - Was leistet mein System für seine Umwelt (Nachbarsysteme, Stakeholder)?
- Diese Stärken hat das Diagramm:
  - präsentiert die Außensicht auf das System
  - geeignet zur Kontextabgrenzung
  - hohes Abstraktionsniveau, einfache Notationsmittel

Wiederholung UML:
Anwendungsfalldiagramm



Spezifiziert einen Anwendungsfall des Systems:
Szenario einer Funktionalität, die einem Benutzer oder einem anderen System angeboten wird.

Akteure die mit einer Aktivität verknüpft sind.

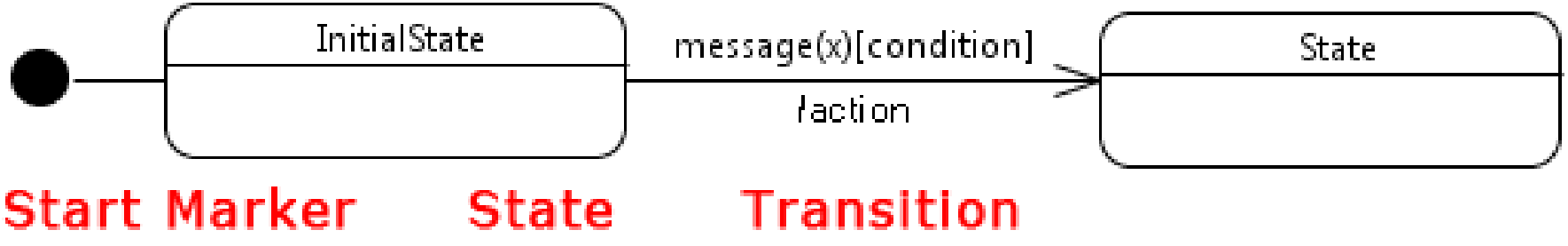
Wiederholung UML: Statechart (Zustandsdiagramm)

- Beschreibung des Verhaltens von Anwendungsfällen
  - Die formale Modellierung von Anwendungsfällen hat folgende Vorteile:
    - Sie sind eindeutig und weniger interpretierbar.
    - Es können Testfälle abgeleitet werden.
- Beschreibung des Verhaltens von Klassen
  - Dem Attribut einer Klasse wird im Allgemeinen ein Datentyp zugeordnet.
  - Wenn der Datentyp eine endliche Menge von gültigen Werten besitzt, dann ergeben sich die verschiedenen Zustände der Klasse aus allen möglichen Kombinationen dieser Zustandsvariablen.

Wiederholung UML: Statechart (Zustandsdiagramm)

Dynamisches Verhalten der einzelnen Komponenten.

Die Eingabe verursacht einen Zustandsübergang und möglicherweise eine Ausgabe.



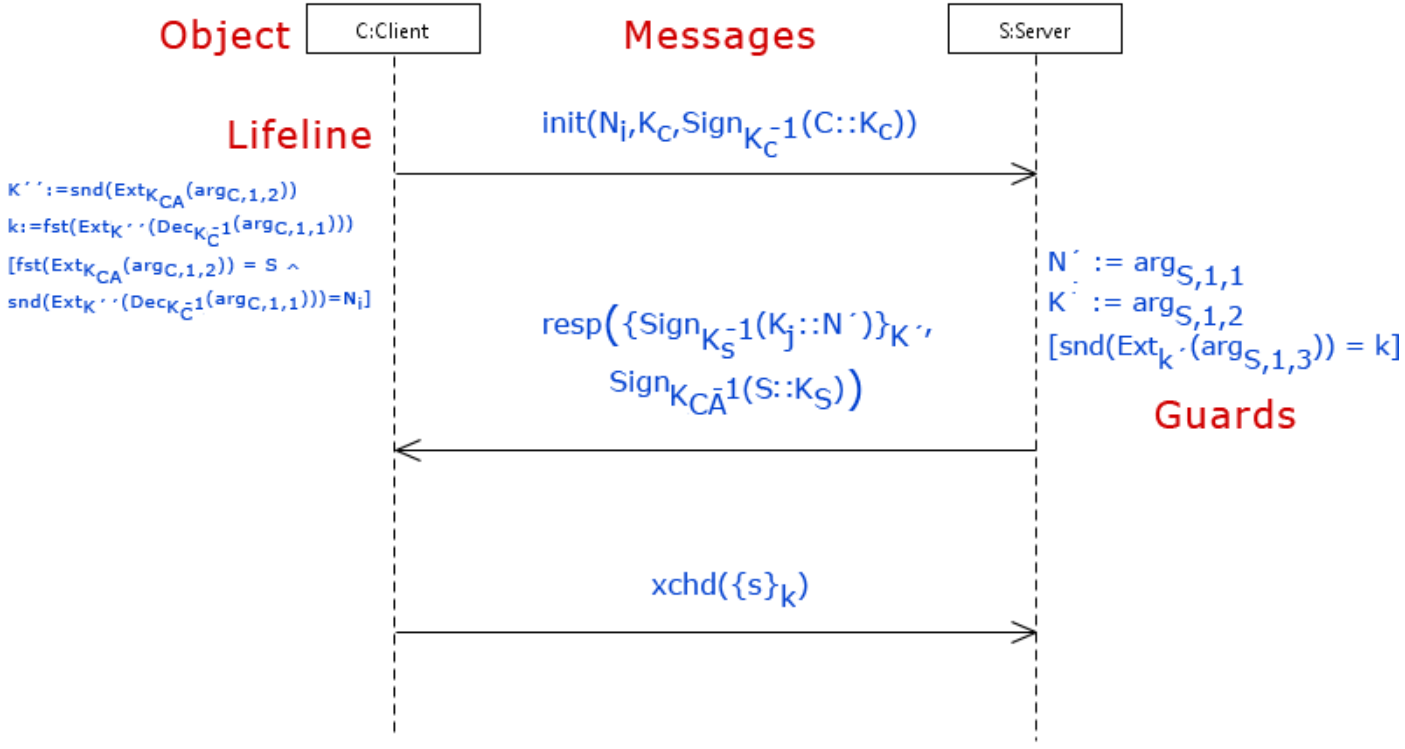
Wiederholung UML:
Sequenzdiagramm

- Diese zentrale Frage beantwortet das Diagramm:
  - Wer tauscht mit wem welche Informationen in welcher Reihenfolge aus?
- Diese Stärken hat das Diagramm:
  - stellt detailliert den Informationsaustausch zwischen Kommunikationspartnern dar
  - sehr präzise Darstellung der zeitlichen Abfolge auch mit Nebenläufigkeiten
  - Schachtelung und Flusssteuerung (Bedingungen, Schleifen, Verzweigungen) möglich

Wiederholung UML:
Sequenzdiagramm

- Häufige Anwendungsfälle
  - Die Abfolge der Nachrichten ist wichtig.
  - Die durch Nachrichten verursachten Zustandsübergänge sind kaum relevant.
  - Die Interaktionen sind kompliziert.
  - Die strukturelle Verbindung zwischen den Kommunikationspartner ist nicht relevant.
  - Es stehen Ablaufdetails im Vordergrund.

Wiederholung UML:
Sequenzdiagramm



Verdeutlicht die Interaktion zwischen Objekten und Komponenten per Nachrichtenaustausch.

Alte Folien (nicht exportieren)

Software-Erstellung: Weitere akute Probleme

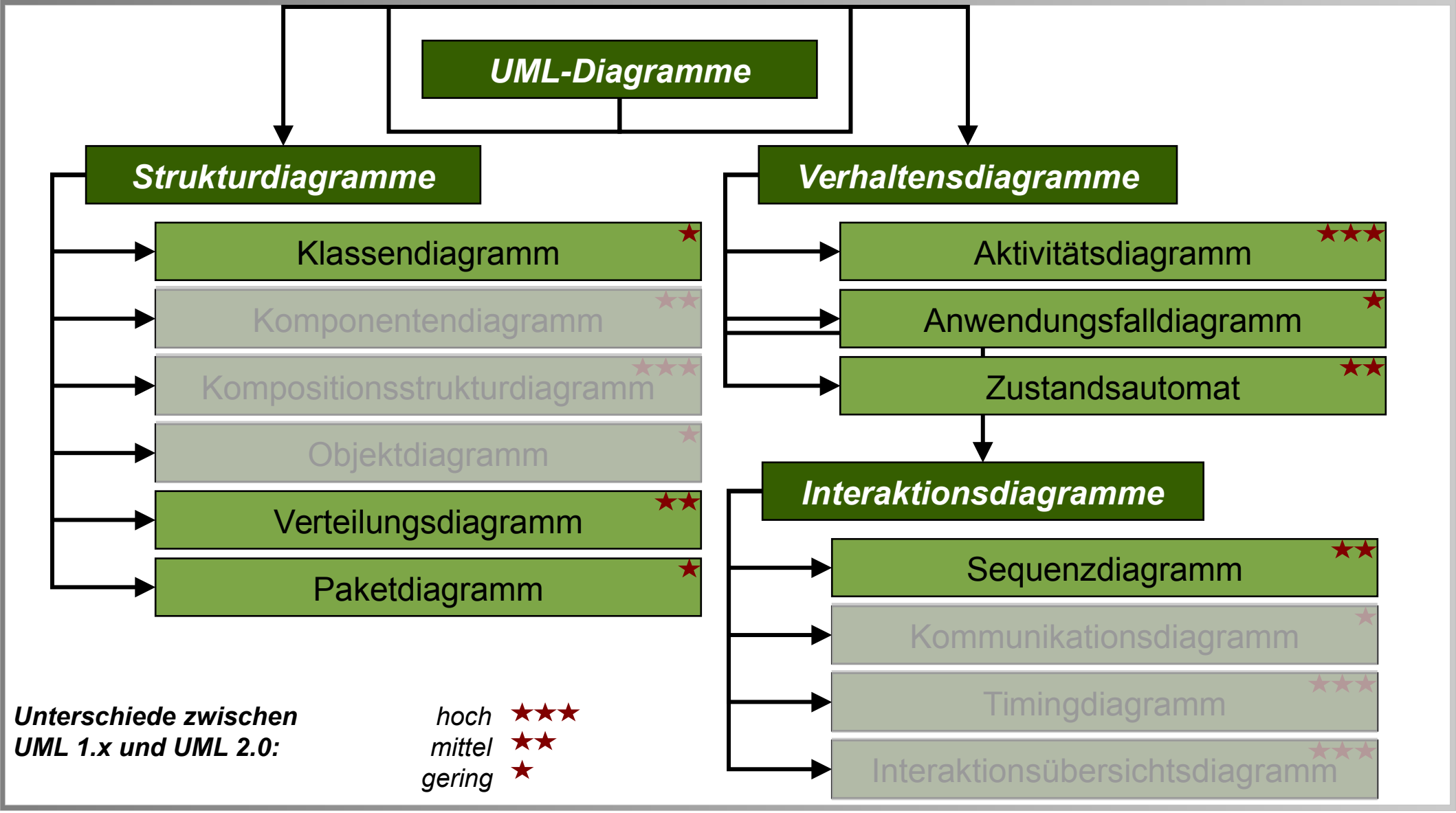
Copy+Paste+Modify-Phänomen:

- **Gleichförmigkeit** häufig übersehen oder schlicht ignoriert.

Descriptor Hell (komplexe Konfigurationsdateien für Deployment):

- **Middleware** weitgehend von technischen Aspekten der Verteilung abstrahiert.
- **Verteilungsflexibilität** mit Zwang zu umfangreicher Konfiguration erkauft.
  - Teil der Einstellungen durch sinnvolle Vorgabewerte abdecken.
  - Deklarative Beschaffenheit legt Umsetzung in Diagrammform nahe.

UMLsec-relevante Diagrammtypen



UMLsec Anmerkungen

- UML2 Vorstellung auf UMLsec relevante Diagramme beschränkt
- UMLsec auf UML 1.5 definiert
  - Beispiele können von UML 2 Abweichen
- UMLsec wurde zum Großteil schon auf UML 2 angehoben (Siehe UMLSec4UML2 Profil), aber um Verwirrung vorzubeugen alle Beispiele in UML 1.5

UML im Schnelldurchlauf
Aktivitätsdiagramm

- Änderungen gegenüber früheren UML-Versionen
  - sind jetzt keine Sonderform der Zustandsdiagramme mehr, sondern basieren auf erweiterten Petri-Netzen
  - damit sind viele Einschränkungen beseitigt:
    - verbesserte Testbarkeit
    - fast automatisch erkennbare Verklemmungsfreiheit
    - bessere Unterstützung paralleler Flüsse
    - Ausführbarkeit fast vollständig möglich
    - mehr Flexibilität in der Modellierung durch das Tokenkonzept

UML im Schnelldurchlauf
Aktivitätsdiagramm stark geändert in UML2

- Tokenkonzept
  - Ein Token (auch: Marke) zeigt an, an welchem Punkt sich der Ablauf gerade befindet.
  - Es können beliebig viele Token unterwegs sein (parallele Abläufe).
  - Token werden graphisch nicht dargestellt, sondern dienen nur der Erklärung der Abläufe.

Wiederholung: Beispiel UML-Spezifikation