

*Vorlesung (WS 2014/15)*  
***Softwarekonstruktion***

**Prof. Dr. Jan Jürjens**

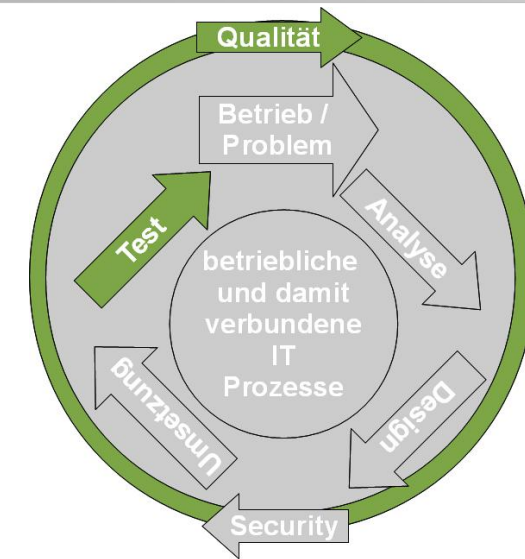
TU Dortmund, Fakultät Informatik, Lehrstuhl XIV

Teil 1.0: Modellbasierte Softwareentwicklung: Einführung

v. 13.10.2014

# 1.0 Modellbasierte Softwareentwicklung: Einführung

- Modellgetriebene SW-Entwicklung
  - Einführung
  - Modellbasierte Softwareentwicklung
  - Object Constraint Language (OCL)
  - Ereignisgesteuerte Prozesskette (EPK)
  - Petrinetze
  - Eclipse Modeling Foundation (EMF)
- Qualitätsmanagement



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

## Literatur:

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

- Kapitel 1-2

# Einleitung: Modellbasierte SE: Einführung

## Kapitel 1:

**Ziel: Vertiefung** der **modellbasierten Softwareentwicklung**, Fortsetzung der elementaren Inhalte (UML-Diagramme) aus Softwaretechnik (SWT)

## Inhalte:

- Modellbasierte Softwareentwicklung: Metamodellierung, Modelltransformation, Object Constraint Language
- Geschäftsprozessmodellierung: Ereignisbasierte Prozessketten (EPKs)
- Ausführungssemantik mit Petrinetzen
- UML-Werkzeugbau: Eclipse Modeling Foundation (EMF)

## Abschnitt 1.0:

- Kurze **Einführung** und **Motivation** für die Inhalte von Kapitel 1.

# 1.0 Modellbasierte Softwareentwicklung: Einführung

Softwarekonstruktion  
WS 2014/15



**1.0  
Modell-  
basierte  
Software-  
entwicklung:  
Einführung**

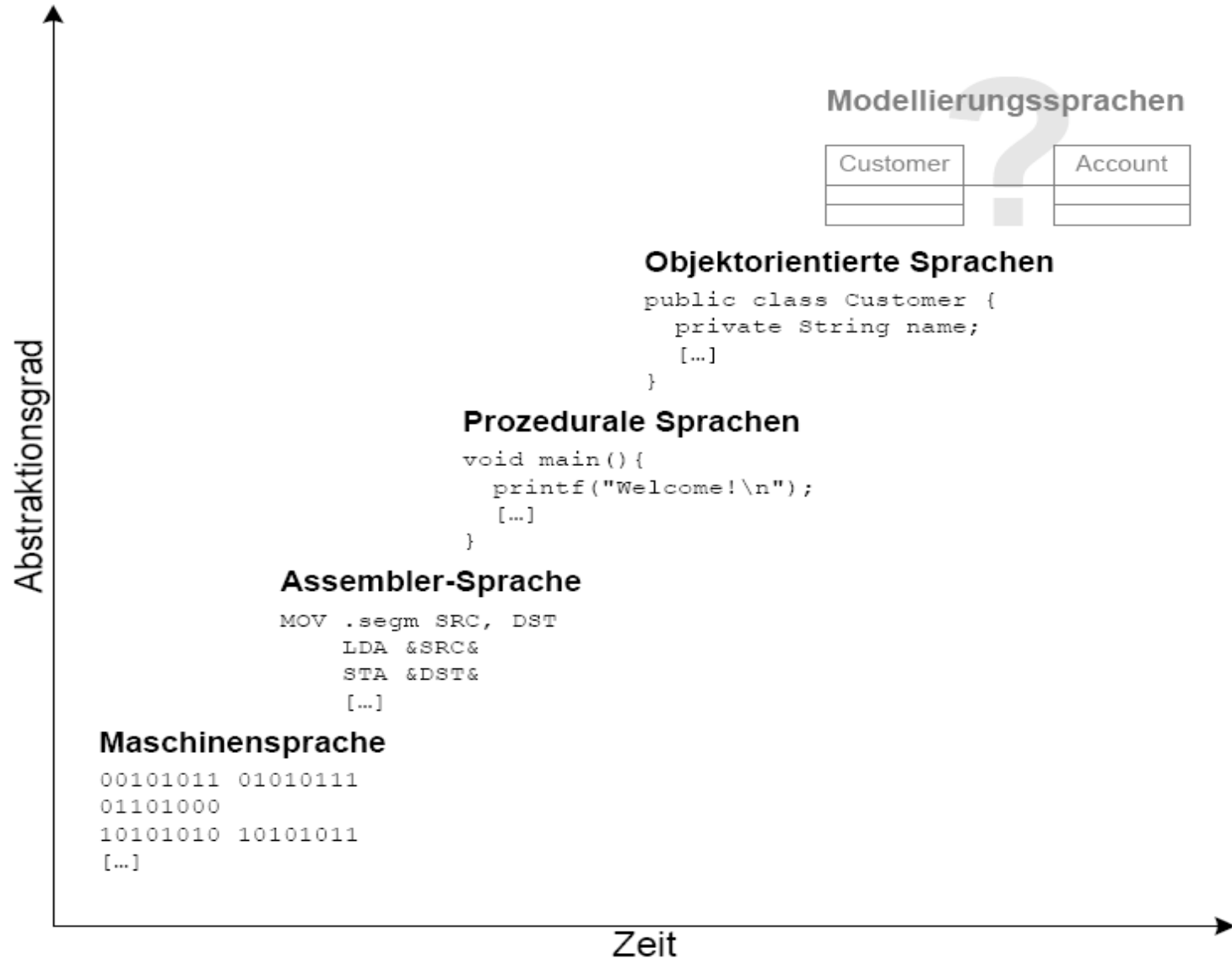


Probleme bei der Softwareentwicklung

Modellbasierte Softwareentwicklung

MDA: Grundlagen und Konzepte

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



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 “New Beetle“ (US) **blockiert gesamte Automatik.**

## 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

### Entwicklungscomplexität

- 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

Ansätze, um **Komplexität zu beherrschen:**

## **Abstraktion:**

- Ausblenden von Detailinformation.
- Einsatz von geeigneten SW-Modellen (**dieses Kapitel !**).

## **Strukturierung / Modularisierung:**

- Aufteilung in klar abgegrenzte Unterstrukturen.
- Partitionierung der Aufgaben (Dekomposition).

## **Methodik und Systematik:**

- Systematisierung des Entwicklungsprozesses.
- Verwendung bewährter Verfahren und Lösungsmuster.
  - Design Pattern (**Abschnitt 1.1 !**),
  - Referenzarchitekturen.



## 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

## 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.

## **Methodischer Bruch** zwischen Analyse, Design und Implementierung:

- Dokumentation nach Beginn Implementierung oft **nicht aktualisiert**.
  - Macht Dokumentation nahezu nutzlos.
- **Erschwert Einarbeitung** neuer Mitarbeiter in späten Phasen.
- **Erhöht Rüstzeiten** in Betriebs- bzw. Wartungsphase um Vielfaches.

## **Fehlende Nachverfolgbarkeit (Traceability):**

- Fehlende Durchgängigkeit für verschiedene Artefakte (Anforderungen, Dokumentation, Code) im Lebenszyklus.
- Besonders in Bezug auf Anforderungen, fehlende
  - **Nachvollziehbarkeit** und
  - **Rückverfolgbarkeit**

- Bedarf nach **höherem Abstraktionsniveau**.
- Potenzial, Gleichförmigkeiten in verdichteten Form zusammenzufassen.
- Forderung nach **durchgängigerer Auswahl der Ausdrucksmittel**.

## → Lösung: Modelle

- **Abstrahieren** und **fokussieren** auf das **Wesentliche**.
- Brücke von **fachlicher Problemwelt** in **technische Lösungswelt**.

# 1.0 Modellbasierte Softwareentwicklung: Einführung

## 1.0 Modell- basierte Software- entwicklung: Einführung



---

Probleme bei der Softwareentwicklung

Modellbasierte Softwareentwicklung

---

MDA: Grundlagen und Konzepte

---

## Kernideen:

- Modell: **Zentrales Artefakt** im SW-Prozess.
  - **Konsequente Nutzung** Lebenszyklus hindurch (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)**  
(aber nicht notwendig).

# Diskussion: Modellbasierte Szenarien

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

# Diskussion: Modellbasierte Szenarien

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

- **Spezifikation**
- **Dokumentation und Kommunikation**
- **Simulieren**
- **Testen**
- **„Programmieren“**



## Spezifikation mit Modellen:

- Verbindliche Spezifikation zwischen **Auftraggeber** und IT-Firma.

## Dokumentation und Kommunikation mit Modellen:

- **Analysemodelle**: Fachexperte und SW-Architekt.
- **Entwurfsmodelle**: SW-Architekt und SW-Entwickler.
- **Implementierungsmodelle**:
  - **Verfeinerung** und **Spezialisierung** des Modells in Entwurfsphase.
  - Vorlage zur Implementierung durch **SW-Entwickler**.
- Generieren von textueller **Dokumentation** aus Modell.

## Simulieren mit Modellen:

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

## Testen mit Modellen:

- Ableiten von Tests (z.B. **Integrations-** und **Abnahmetests**) aus fachlichem Modell.
- **Automatische Generierung** von technischen Testfällen (JUnit).

# Modellbasierte Szenarien: „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  
(z.B. UI, Validierung und Persistenz aus XML-Datei).
- Generatoren für speziellen **Projektzweck**  
(z.B. Kapselung kundenspezifischer Besonderheiten, Produktlinien).

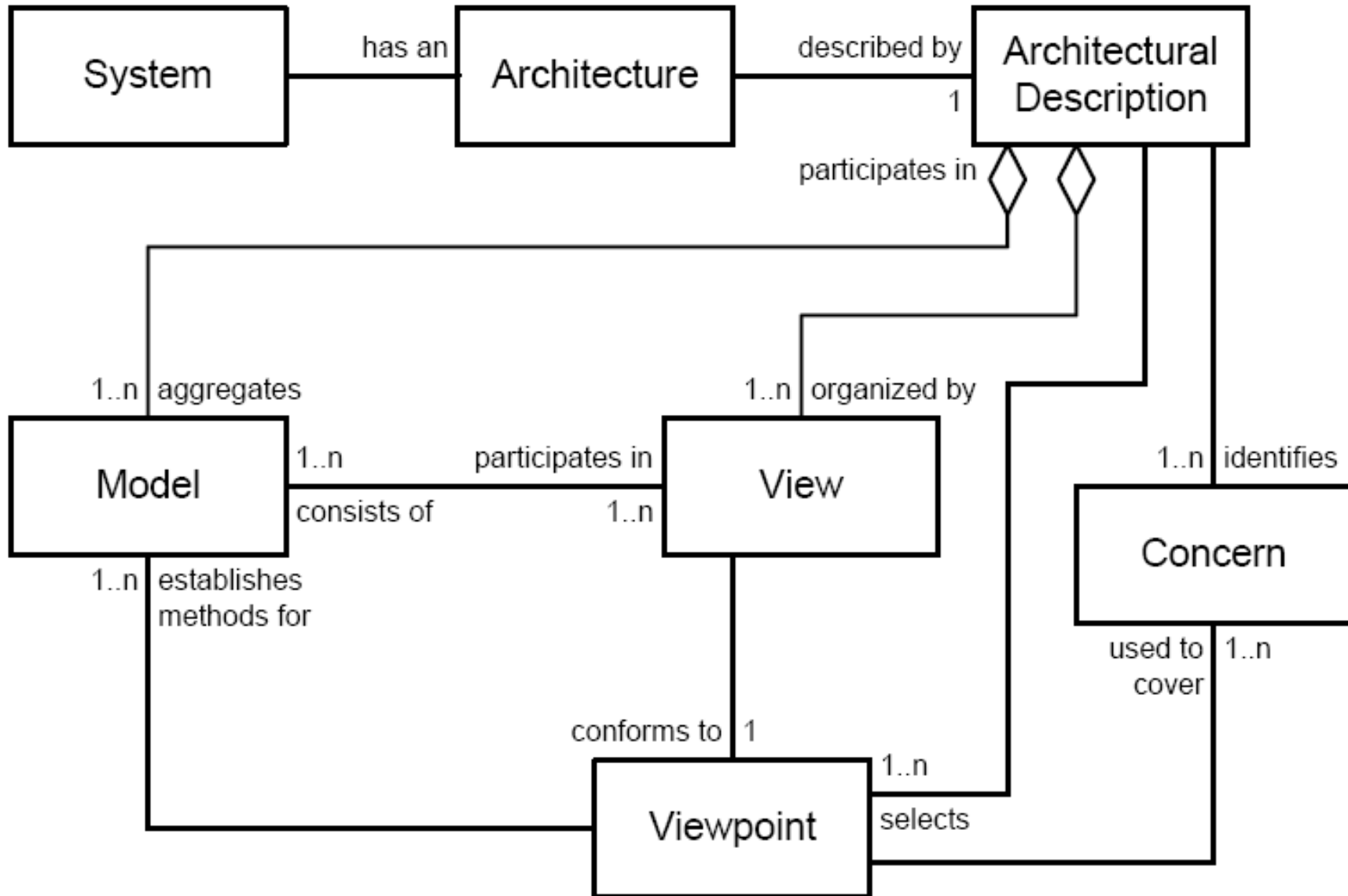
## Modell:

- Abbildung: Welt → Diskrete Struktur.
- Vereinfachende Beschreibung der Realität.

## Einige Aspekte, die modelliert werden:

- Struktur.
- Beziehungen.
- Verhalten.

Grenzen zwischen **Modell** und **Code** fließend:  
kann Code als Modell auffassen.

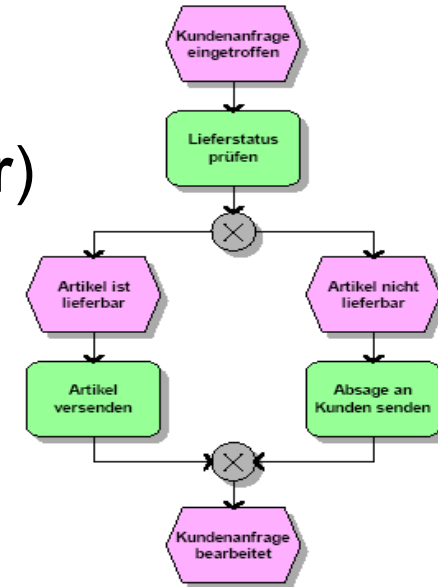


**Welche Arten von Modellen im Bereich Software Engineering kennen Sie bereits ?**

## Beispiele: Modelle im Software Engineering:

### Grafische Modelle (Schwerpunkt: menschlicher Nutzer)

- Unified Modelling Language
- Ereignisgesteuerte Prozessketten (EPK)  
(z.B. für Geschäftsprozesse)
- Petrinetze



### Modelle auf Textbasis:

- XML-Schema (Schwerpunkt: **maschinelle Verarbeitung**)
- Matlab Code

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" >
  <xs:element name="auftrag">
    <xs:complexType>
      <xs:all>
        <xs:element name="kundennummer" type="xs:string"/>
        <xs:element name="auftragsdatum" type="xs:string"/>
        <xs:element name="ausführungsdatum" type="xs:string"/>
        <xs:element name="auftragsposition">
          <xs:complexType><xs:sequence base="xs:string"
            ref="auftragspositiontype"/></xs:sequence>
        </xs:element>
        <xs:element name="kundenanschrift">
          <xs:complexType><xs:sequence base="xs:string"
            ref="kundenanschrifttype"/></xs:sequence>
        </xs:element>
        <xs:element name="rechnungsanschrift">
          <xs:complexType><xs:sequence base="xs:string"
            ref="rechnungsanschrifttype"/></xs:sequence>
        </xs:element>
        <xs:element name="installationsanschrift">
          <xs:complexType><xs:sequence base="xs:string"
            ref="installationsanschrifttype"/></xs:sequence>
        </xs:element>
      </xs:all>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

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

**Semantik:** Bedeutung, die durch Modell ausgedrückt werden soll, z.B.:

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



Modellverarbeitung in modellbasierter Entwicklung:

Werkzeuge benötigen 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:** kein einheitliches Format

- UML: Object Constraint Language (OCL).
- Alternativen: Petrinetze (s. **Abschnitt 1.4 !**), Abstract State Machines, Logik erster Stufe, Temporale Logik, ...

# 1.0 Modellbasierte Softwareentwicklung: Einführung

Softwarekonstruktion  
WS 2014/15



**1.0  
Modell-  
basierte  
Software-  
entwicklung:  
Einführung**



---

Probleme bei der Softwareentwicklung

---

Modellbasierte Softwareentwicklung

---

MDA: Grundlagen und Konzepte

**Ein** konkreter Ansatz für modellbasierte Entwicklung.

**Spezifikation der Software** unabhängig von technischer Umsetzung.

**Verschiedene Abstraktionslevel** für Modelle:

- **Plattform-unabhängig.**
- **Plattform-spezifisch.**

Übergang von abstrakten zu technologiespezifischen Modellen:

- **Vollautomatisiert:** Verwendung von Transformationswerkzeugen.
- **Beschreibung der Transformationen** in Transformationsbeschreibungen.

# Zur Diskussion: Stärken von MDA ?

Welche der folgenden möglichen Ziele könnten Ihrer Meinung nach **Stärken** von MDA sein ?

- Domänenorientiertes Design
- Effiziente Softwareentwicklung
- Systemintegration & Interoperabilität
- Portierbarkeit

**Abstimmung ?**

# Ziele der MDA: Domänenorientiertes Design

**Problem 1: Wertvolles Wissen über Kerngeschäftsprozesse** oft implizit in proprietären Altsystemen bzw. in Köpfen der Entwickler.

**Problem 2: Bedürfnisse der Kunden:**

- **Hohe Flexibilität / Agilität** unterliegender Geschäftsprozesse.
- **Keine unnötige Beschränkung** durch technologische Vorgaben.

**Lösung: Orientierung an Fachlichkeit** innerhalb des SE-Zyklus („Domänenorientierung“).

# Ziele der MDA: Domänenorientiertes Design

## 1) Plattform-unabhängige Modellierung der unterliegenden Prozesse:

- Pflege und Weiterentwicklung der Anwendungslogik.
- Dokumentation und Evolution der Prozesse.
- Integration der Modelle in Transformationskette hin zur Anwendung.

→ **Gegen Degeneration von Dokumentation.**

## 2) Domänenspezifische Modelle:

- Ziel: **Reduzierung** der **Time-to-Market** ohne Budget-Erhöhung.
- **Wissensvorsprung** innerhalb fachlichen Domänenwissens.

# Ziele der MDA: Effiziente Softwareentwicklung

## Automatisierung der Anwendungserstellung:

- Beschleunigt Software-Prozesse, verringert Aufwand.
- Steigert Softwarequalität.
  - z.B. Verwendung von Referenzarchitekturen / Patterns.

## Technologien gekapselt:

- Expertenwissen via **Transformationsvorschriften** wiederverwenden.
- **Optimale** Nutzung vorhandener Ressourcen (Wiederverwendung).
- Beherrschung **systemimmanenter Komplexität**
  - Prinzip: Teile-und-herrsche.

## Trennung fachlicher Konzepte von konkreter technologischer Repräsentation:

- Erleichtert Bildung von **Schnittstellen**.

## Verwendung offener Standards:

- Verringert Risiko von Vendor-Lock-Ins.
  - Hilft bestehende Anwendungen über Technologiezyklen zu retten.
- Erleichtert **Wiederverwendung** und steigert **Produktivität**.



**Systematische Abstraktion** von technischen Aspekten erleichtert:

- Migration von Applikationen auf neue Versionen benutzter Technologien.
- Portierung auf andere Zielumgebungen.
- Anwendungsentwicklung für mehr als eine Zielplattform.

# MDA: Verschiedene Abstraktionsebenen für Modelle

## Welche Abstraktionsebene erfüllt welches Ziel ?

Anforderungen an System und Umwelt.

Platform  
Independent  
Model (PIM)

Spezifikation von Struktur und Verhaltens  
des Systems unabhängig von  
Hardware- & Softwareplattformen.

Platform  
Specific  
Model  
(PSM)

Alle Informationen um System erzeugen  
und in Betrieb nehmen.

Computation  
Independent  
Model (CIM)

# MDA: Verschiedene Abstraktionsebenen für Modelle

## Welche Abstraktionsebene erfüllt welches Ziel ?

Anforderungen an System und Umwelt.

Spezifikation von Struktur und Verhaltens des Systems unabhängig von Hardware- & Softwareplattformen.

Alle Informationen um System erzeugen und in Betrieb nehmen.

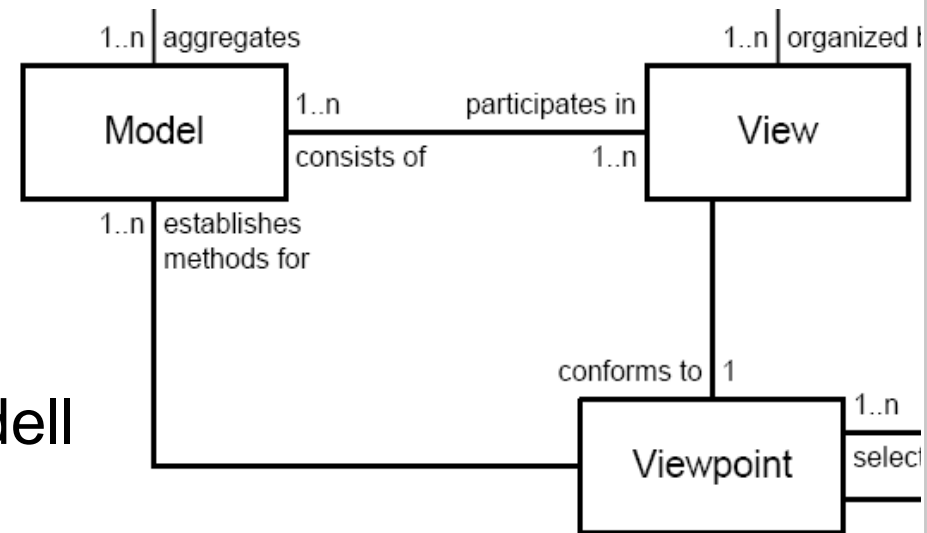
Platform Independent Model (PIM)

Platform Specific Model (PSM)

Computation Independent Model (CIM)

## Unified Modeling Language (UML):

- **Modellierung, Dokumentation, Spezifizierung und Visualisierung von komplexen Softwaresystemen.**
- **Standard** der Object Management Group (OMG).
- **Unterschiedliche Modellierungskonzepte** auf einheitlicher Basis:
  - 14 Diagrammtypen in Version 2.4.1
- **Modell vs. Diagramm:**
  - UML-Modell besteht aus einem oder mehreren Diagrammen.
  - Ein Diagramm entspricht einer bestimmten Sicht („View“) auf Modell (vgl. IEEE 1471).



# UML: Was Sie von SWT wiederholen sollten

Was Sie sich für dieses Kapitel noch einmal anschauen sollten, um dahinterliegende **Metamodellierung** zu verstehen:

- **UML-Klassendiagramme** (Generalisierung, Assoziationen, Komposition, Aggregation, Multiplizität,...)
- **UML-Aktivitätsdiagramme** (Aktivitäten, Aktionen, Verzweigungsknoten, Verbindungsknoten, Verteilungsknoten, Kontrollflüsse,...)

Was auch hilfreich ist zu kennen:

- Weitere **Struktur-** und **Verhaltensdiagramme** der UML (z.B. Objekt-, Sequenz-, Zustandsdiagramme)

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

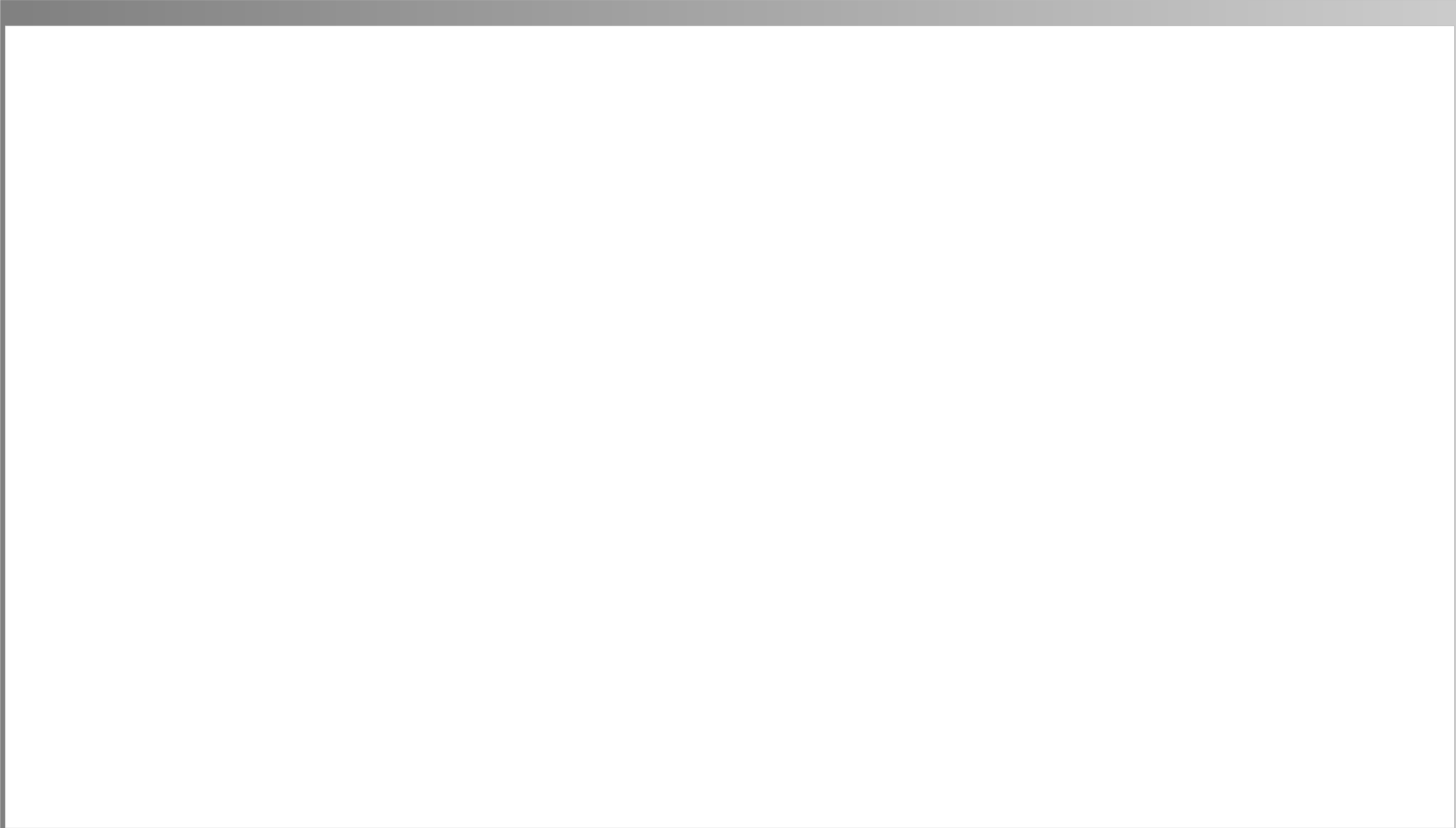
- Aktuelle Herausforderungen
- Modellbasierte Softwareentwicklung
- Modellbegriff und SW-Modelle
- Semantik von Modellen
- MDA

**Nächster Abschnitt:**

Zentrale Techniken der modellbasierten Softwareentwicklung.

# Anhang (Weitere Informationen und Beispiele zum Selbststudium)

Softwarekonstruktion  
WS 2014/15



- **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 größtmäßig **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)<sup>1</sup>.
  - Strukturierte Design (SD)<sup>2</sup>.

<sup>1</sup>Tom DeMarco: Structured Analysis and System Specification. Yourdon Press, 1978.

<sup>2</sup>Edward Yourdon und Larry L. Constantine: Structured Design – Fundamentals of a Discipline of Computer Program and Systems Design. Prentice Hall, 1979.

- **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.

- **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)

## Divergenz der Änderungszyklen:

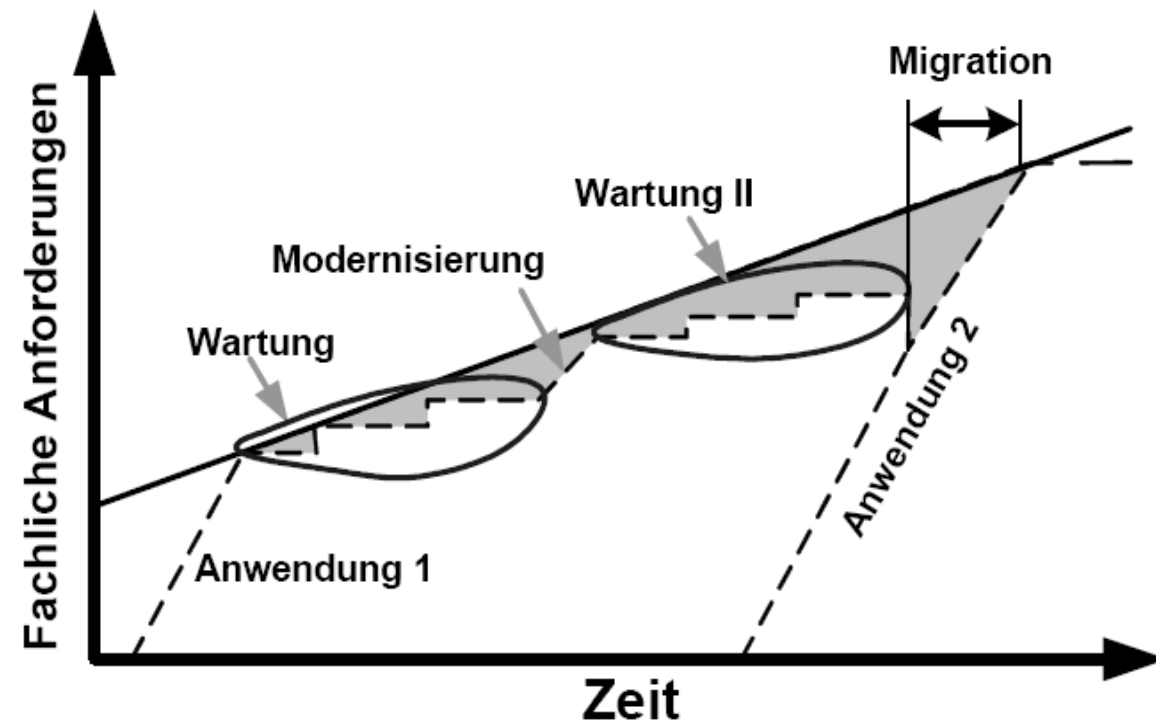
- **Änderungszyklus** der Technologie vs. Änderungen fachlicher Prozesse.
- **Technische und fachliche Belange** der Entwicklung trennen.
  - Besserer **Investitionsschutz** der Entwicklungsarbeit.
  - Verlängerte **Lebensdauer** der Fachkonzepte.
  - Größere **Flexibilität** und bessere Reaktionsfähigkeit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

## 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.





- Welche **akuten Probleme** der Softwareentwicklung lassen sich mit den **Zielen der MDA** verbinden?

Dominierung der Fachlichkeit

Divergenz der Änderungszyklen

Methodischer Bruch zwi. Analyse,  
Design und Implementierung

Middleware-Babel

Descriptor Hell

Konservierung  
der Fachlichkeit

Systemintegration und  
Interoperabilität

Fachlichkeit

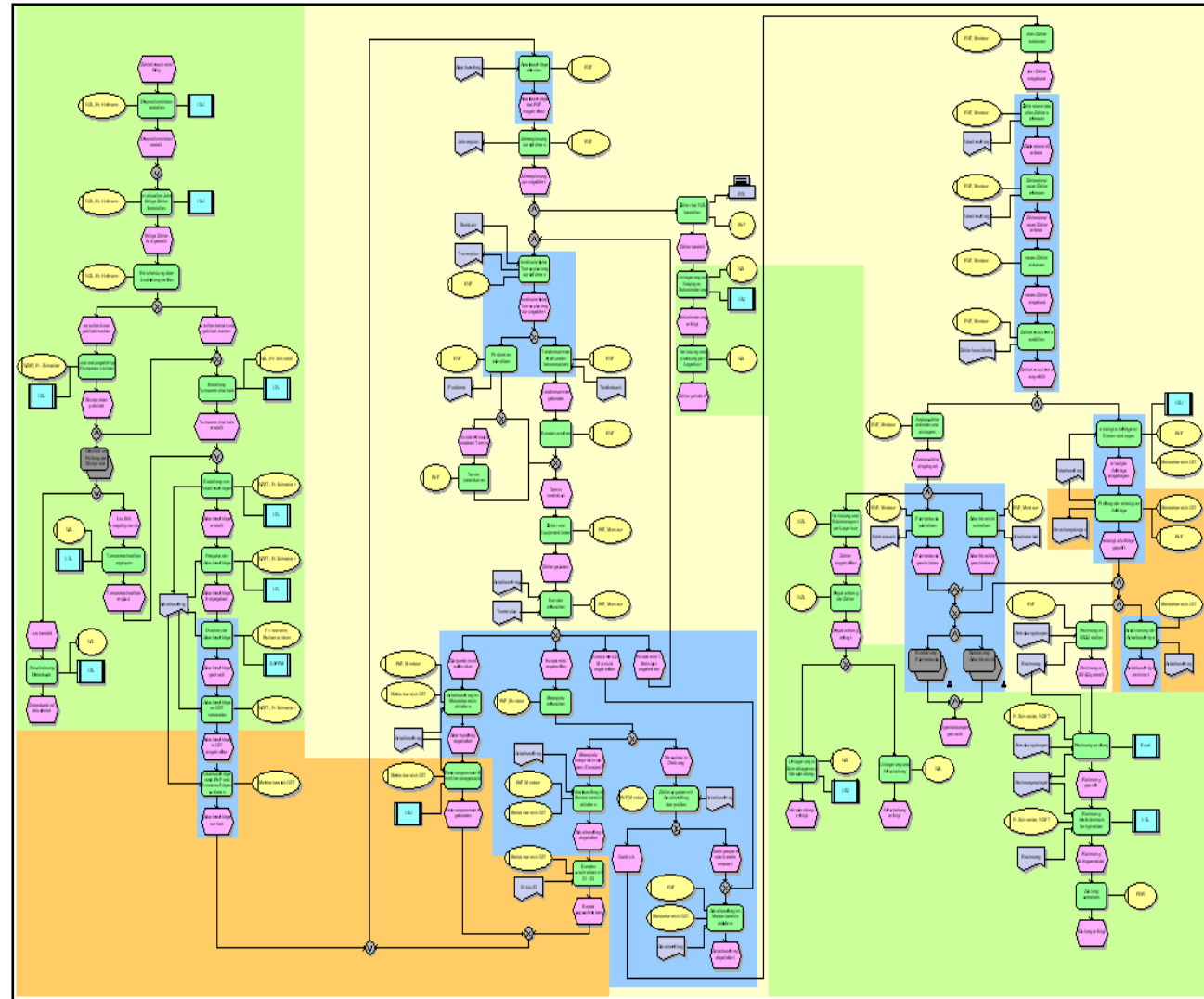
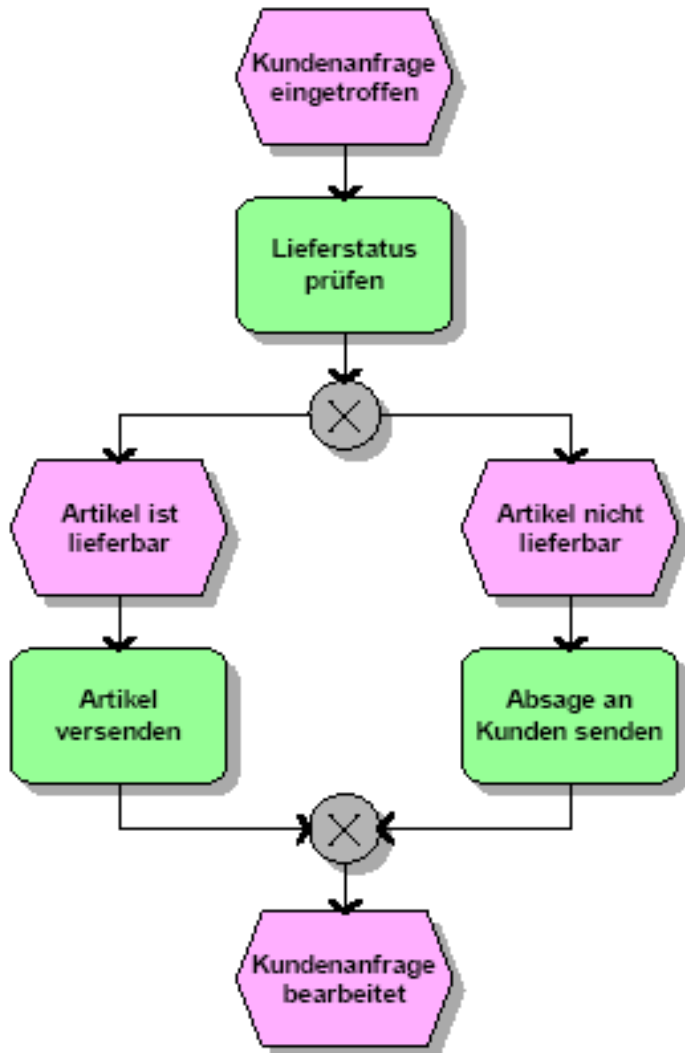
Portierbarkeit

- **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.

```
<?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="adresstype"/></xs:sequence></xs:complexType>
        </xs:element>
        <xs:element name="rechnungsanschrift">
          <xs:complexType><xs:sequence><xs:element ref="adresstype"/></xs:sequence></xs:complexType>
        </xs:element>
        <xs:element name="installationsanschrift">
          <xs:complexType><xs:sequence maxOccurs="unbounded"><xs:element ref="adresstype"/></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="adresstype">
    <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>
```

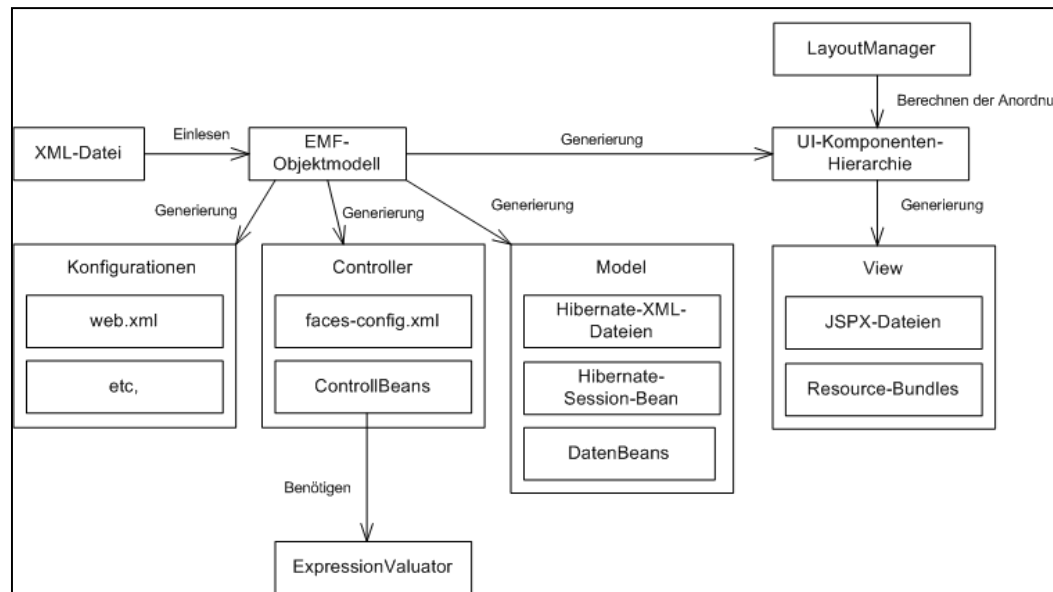
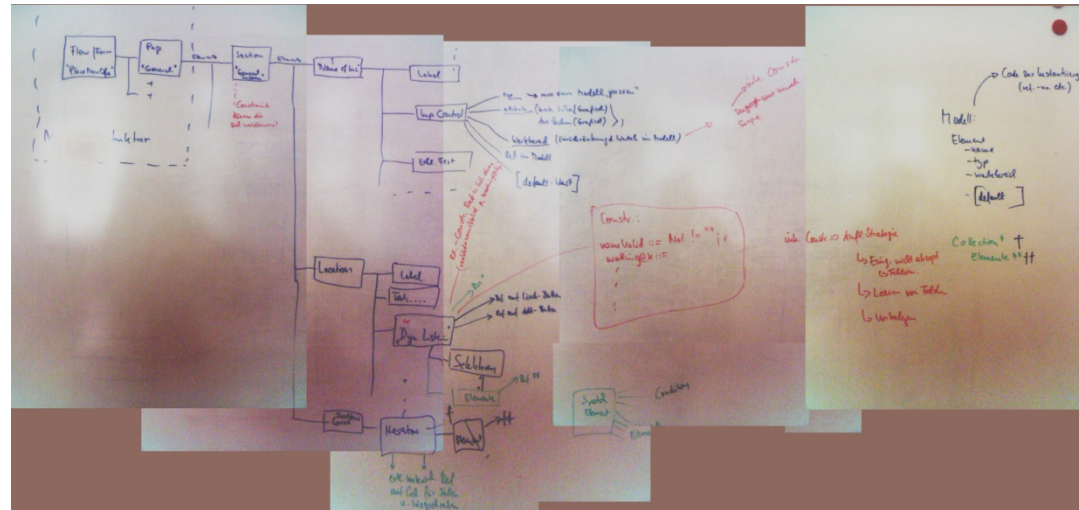
## Ereignisgesteuerte Prozessketten:

- 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.



## 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.



### System:

- Aus Teilen **zusammengesetztes und strukturiertes Ganzes.**
- Hat Funktion und erfüllt Zweck.

### Architecture:

- Organisation der Teile des Systems und deren Verhalten.
- **Beziehungen unter einzelnen Komponenten.**
  - Beziehungen an Bedingungen knüpfbar.
    - Erfüllung zur Erreichung des Systemzwecks.

### Architectural Description:

- Besteht aus **Menge aller Artefakte:**
  - Notwendig zur Beschreibung einer Architektur.
- Typischerweise aus Modellen des Systems und unterstützender Dokumentation.



### Concern:

- **Spezifische Anforderungen** oder Bedingungen.
  - Erfüllen, um erfolgreiche Erfüllung des Systemzwecks zu gewährleisten.
- Als **Aspekt** übersetzt.

### Model:

- Beschreibt oder **spezifiziert System** zu bestimmten Zweck.
- Erfasst **relevante Aspekte**.
  - Struktur, Verhalten, Funktion und Umwelt des Systems.

### Viewpoint:

- Beschreiben Elemente und Konstrukte.  
→ Ermöglichen **Erstellung des Modells** zur Beschreibung von Systemaspekten.
- **Regeln, Techniken bzw. Methoden** sowie Notationsmittel zur Erstellung eines Modells.

### View:

- Kohärente **Menge von Modellen**:
  - Beschreibt bestimmte Aspekte eines Systems.
  - Verbirgt irrelevante Aspekte.
- Als Sicht auf System bezeichnet.
- Mit **Regeln und Notation** des entsprechenden Viewpoints dargestellt.

### Metamodel:

- Modelle mit Menge von Elementen erstellbar.
- Enthält:
  - **Regeln** bei Erstellung des Modells beachten (abstrakte Syntax).
  - Bedeutung der Elemente und Elementkonstellationen (Semantik).

### Profile:

- **Leichtgewichtige Erweiterung** eines Metamodells.
- Stellt **neue Modellelemente** zur Verfügung oder erweitert Semantik oder Syntax bestehender Elemente.
- **Verschärfen Bedingungen**, die an Elemente der Modelle gestellt werden.
  - Regeln eines **Design-by-Contracts** [DBC] unterordnen.

### Domain:

- Abgrenzbares, kohärentes Wissensgebiet.
- Im Metamodell: Konzepte einer Domäne in Bezug gesetzt und beschrieben.
- **Domänenspezifische Sprache:** Sprachdefinierender Charakter, Metamodell und Profil.

### Application:

- Zu erstellendes Stück Software, umfasst **zu entwickelnde Funktionalität**.
- System aus einer oder mehreren Anwendungen zusammensetzbar.  
→ Auf einer oder mehreren Plattformen ausführbar.

### Plattform:

- **Ausführungsumgebung** für Anwendung.
- Zugriff auf Funktionalität über Schnittstellen. → Kenntnis über Implementierung nicht nötig.
- Bsp.: Plattformen im Bereich Betriebssysteme (Unix, Windows, OS/390).
- Bsp.: Programmiersprachen (C++, Java, C#).
- Bsp.: Middleware (CORBA, Java EE, .NET).
- Setzen in Art eines **Plattform-Stacks** aufeinander auf.
- Hardware eines Computers bildet Plattform für Betriebssystem.
- **Betriebssystem:** Plattform für einzelne Anwendungen usw.

### **Computation Independent Model (CIM):**

- Liefert Sicht auf Gesamtsystem unabhängig von Implementierung.
- Modell im Vokabular seiner Domäne beschrieben.
- **Betont Anforderungen an System** und seine Umwelt.
- Synonyme: Business Model, Domain Model.

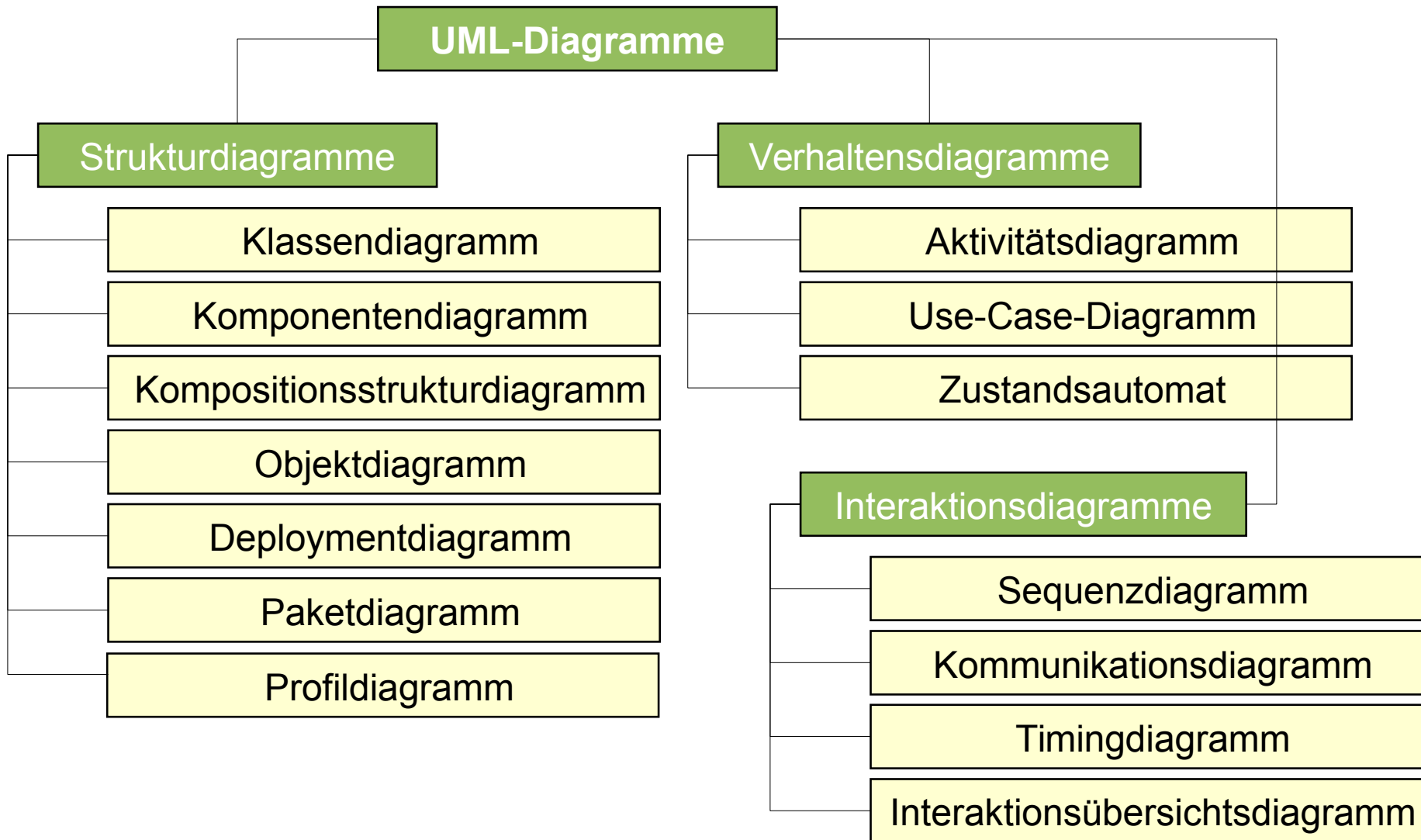
### **Platform Independent Model (PIM):**

- Beschreibt **formale Struktur und Funktionalität** eines Systems.
- Unabhängig von implementierungstechnischen Details. → Abstrahiert von zugrunde liegenden Plattform.

### **Platform Specific Model (PSM):**

- **PIM mit plattformabhängigen Informationen.**
- Zur Implementierung: Falls es alle Informationen zur Konstruktion und Betrieb eines Systems liefert.  
→ Modelle auch direkt ausführbar (executable models).

# 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

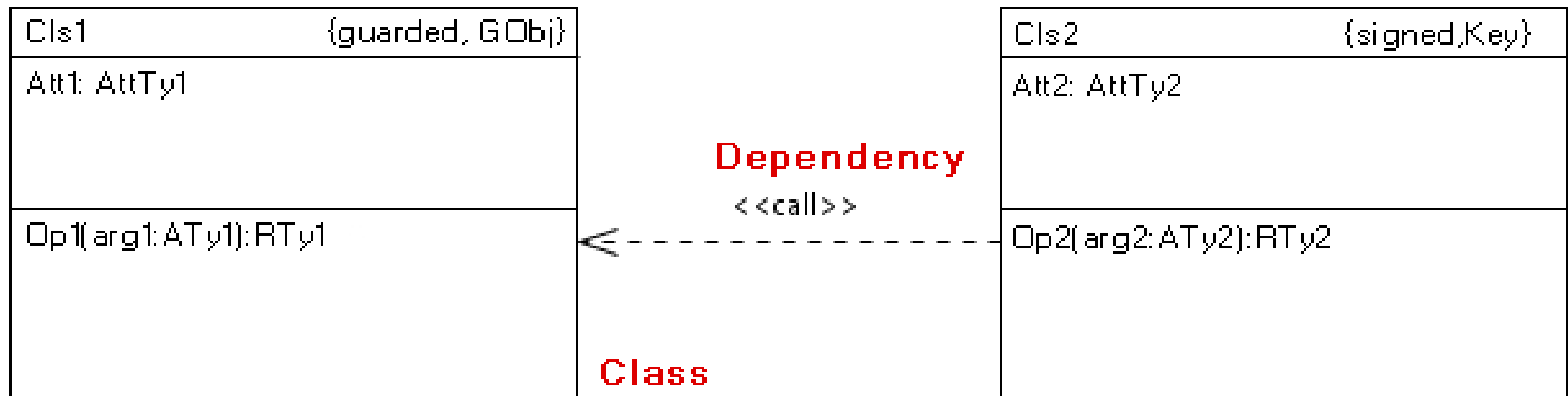
**Verteilungsdiagramm:** Physikalische **Umgebung**

**Paket/Subsystem:** **Fasst** Diagramme eines Systemteils **zusammen**

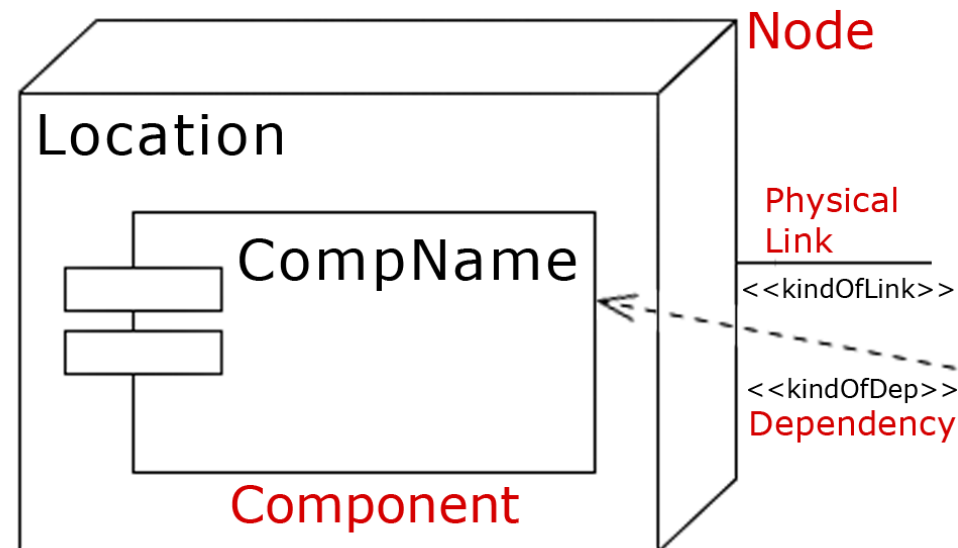


## Klassenstruktur des Systems.

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

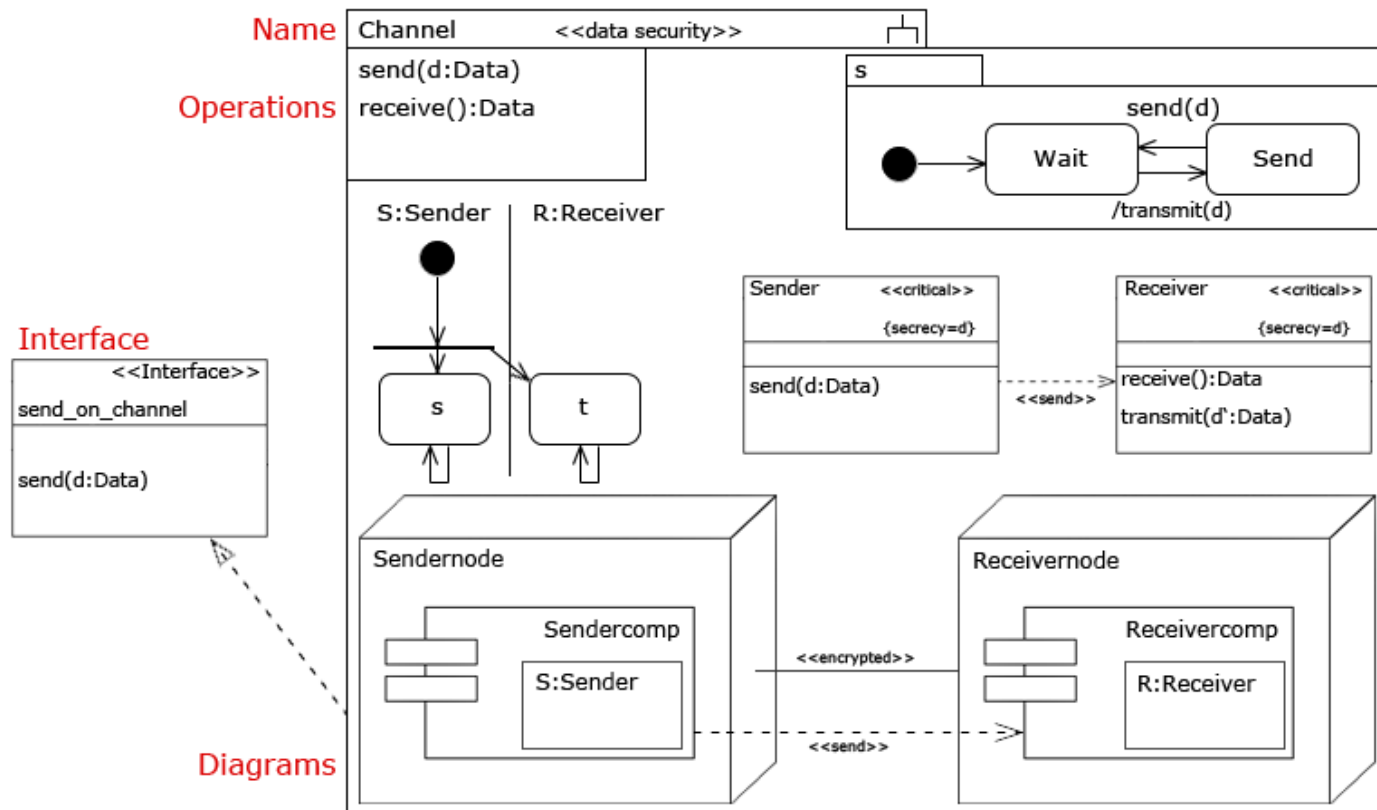


- Problem:
  - Es soll die Verteilung der Software des Systems auf die Hardware beschrieben werden.
- Diese zentrale Frage beantwortet das Diagramm:
  - Wie sieht das Einsatzumfeld (Hardware, Server, Datenbanken etc.) des Systems aus?
  - Wie werden die Komponenten zur Laufzeit wohin verteilt?



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

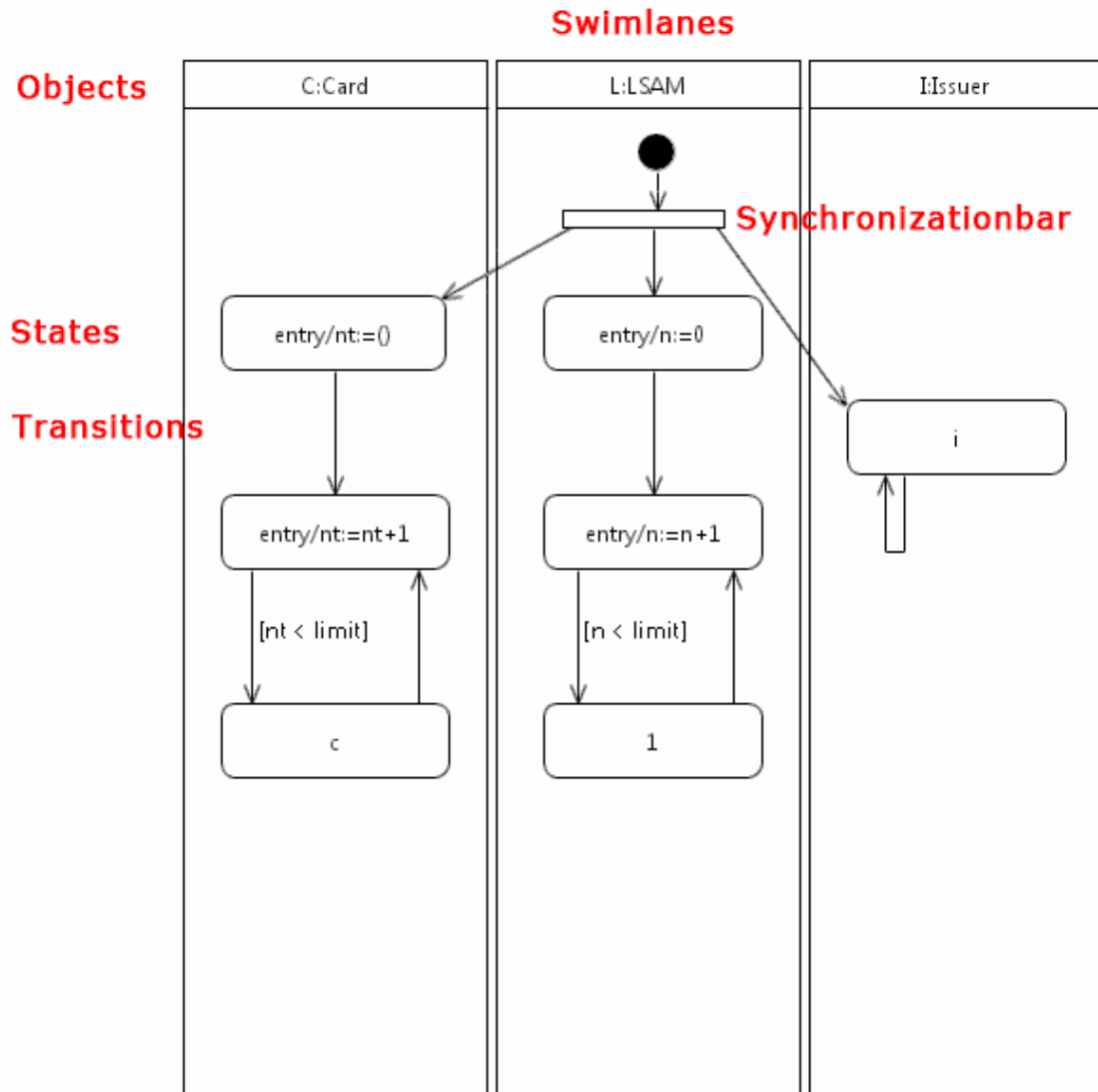
- Problem:
  - Große Softwaresysteme mit mehreren 100 Klassen lassen sich nicht mehr von einer Person überblicken.
- Diese zentrale Frage beantwortet das Diagramm:
  - Wie kann ich mein Modell so schneiden, dass ich den Überblick bewahre?
- Diese Stärken hat das Diagramm:
  - organisiert das Systemmodell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



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

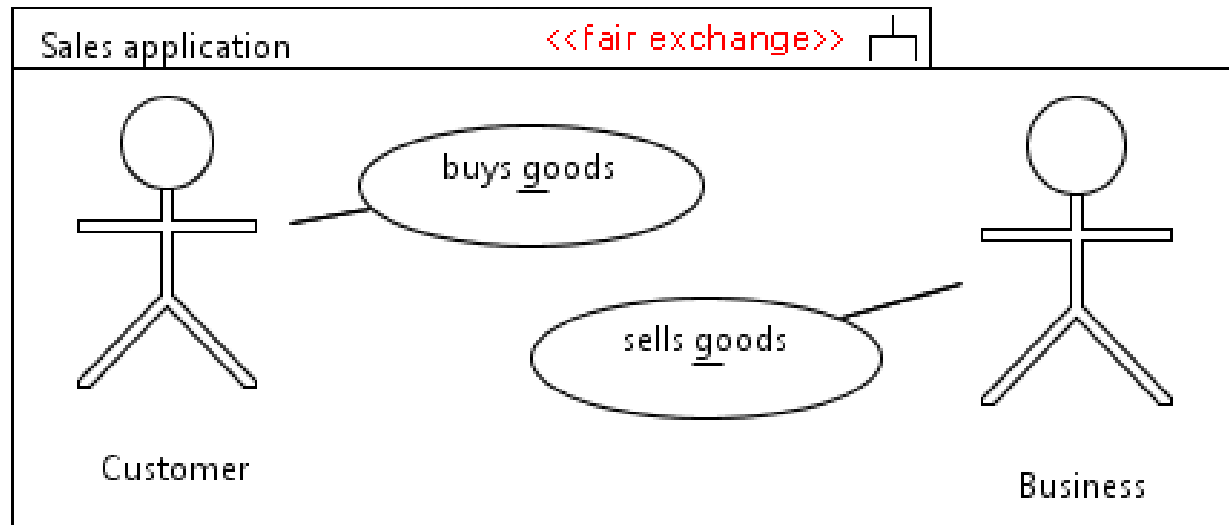
- Problem:
  - Es sollen Abläufe, z.B. Geschäftsprozesse, modelliert werden. Im Vordergrund steht dabei eine Aufgabe, die in Einzelschritte zerlegt werden soll.
  - Es sollen Details eines Anwendungsfalles festgelegt werden.
- Diese zentrale Frage beantwortet das Diagramm:
  - Wie realisiert mein System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

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



- 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





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

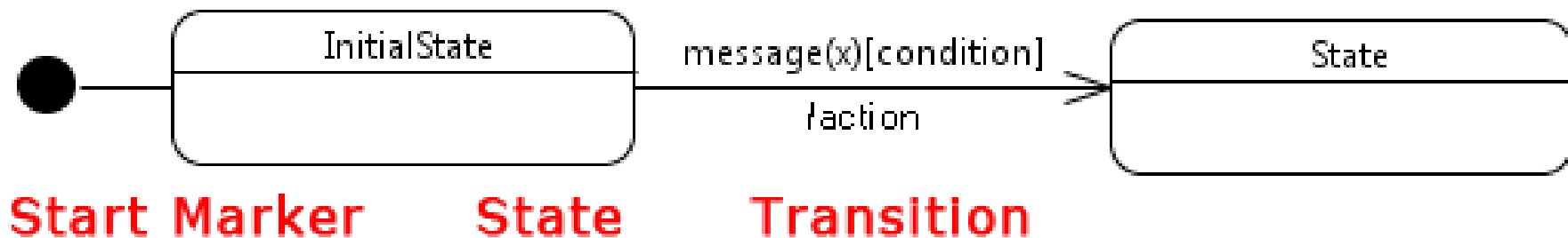
Akteure die mit einer Aktivität verknüpft sind.

- 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.

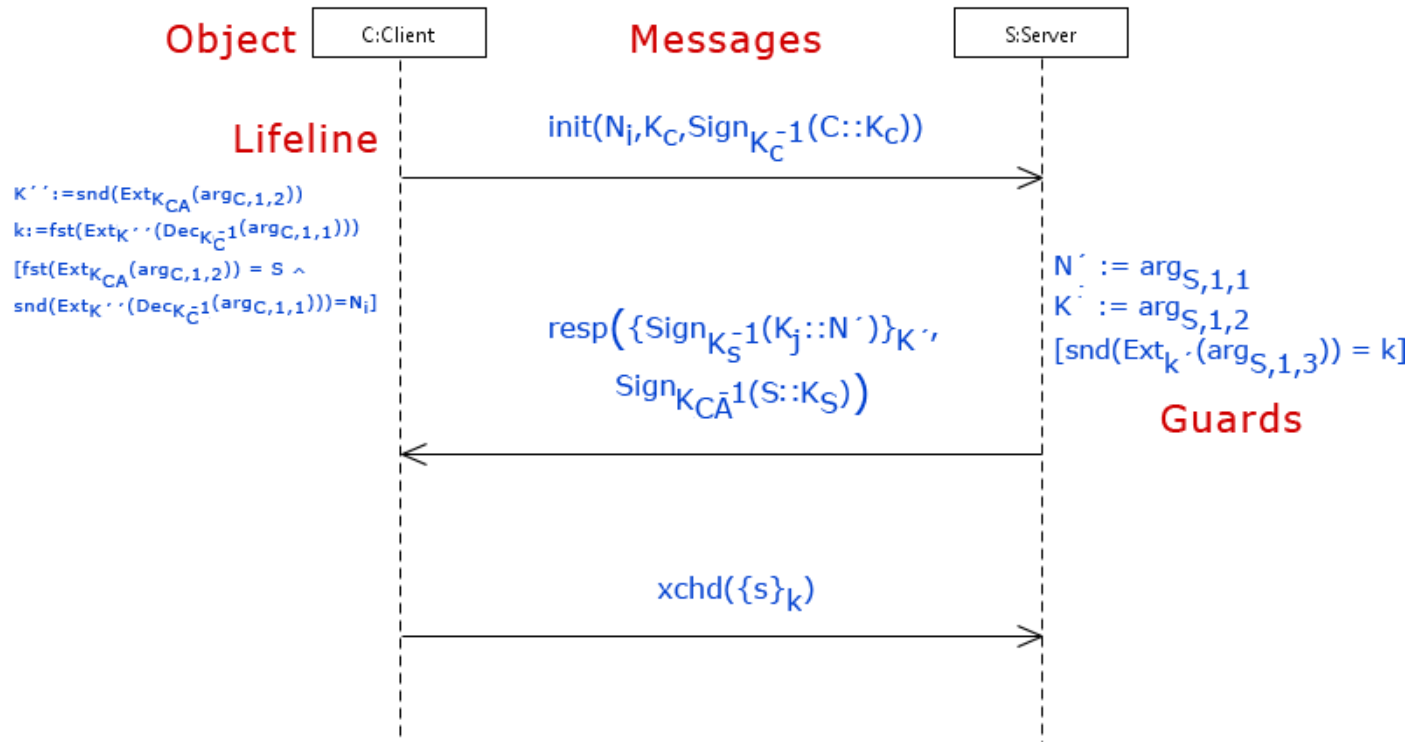
Die Eingabe verursacht einen Zustandsübergang  
und möglicherweise eine Ausgabe.



- 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

- 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 Kommunikationspartnern ist nicht relevant.
  - Es stehen Ablaufdetails im Vordergrund.

# Wiederholung UML: Sequenzdiagramm



Verdeutlicht die **Interaktion** zwischen Objekten und Komponenten per **Nachrichtenaustausch**.

- [BRJ01] The Unified Modeling Language; Booch, Rumbaugh Jacobsen; Addison-Wesley; 2001
- [Beziv01] Towards a Precise Definition of the OMG/MDA Framework. Bézivin, J. and O. Gerbé. ASE'01 - Automated Software Engineering. 2001. San Diego, USA.
- [BHK04] Softwareentwicklung mit UML2; M.Born, E. Holz, O.Kath; Addison-Wesely, 2004
- [BBG05] Model-Driven Software Development; S.Beydeda, M.Book, V.Gruhn; Springer-Verlag 2005
- [CKE00] Generative Programming: Methods, Tools, and Applications; K.Czarnecki, U.Eisenecker; Pearson 2000