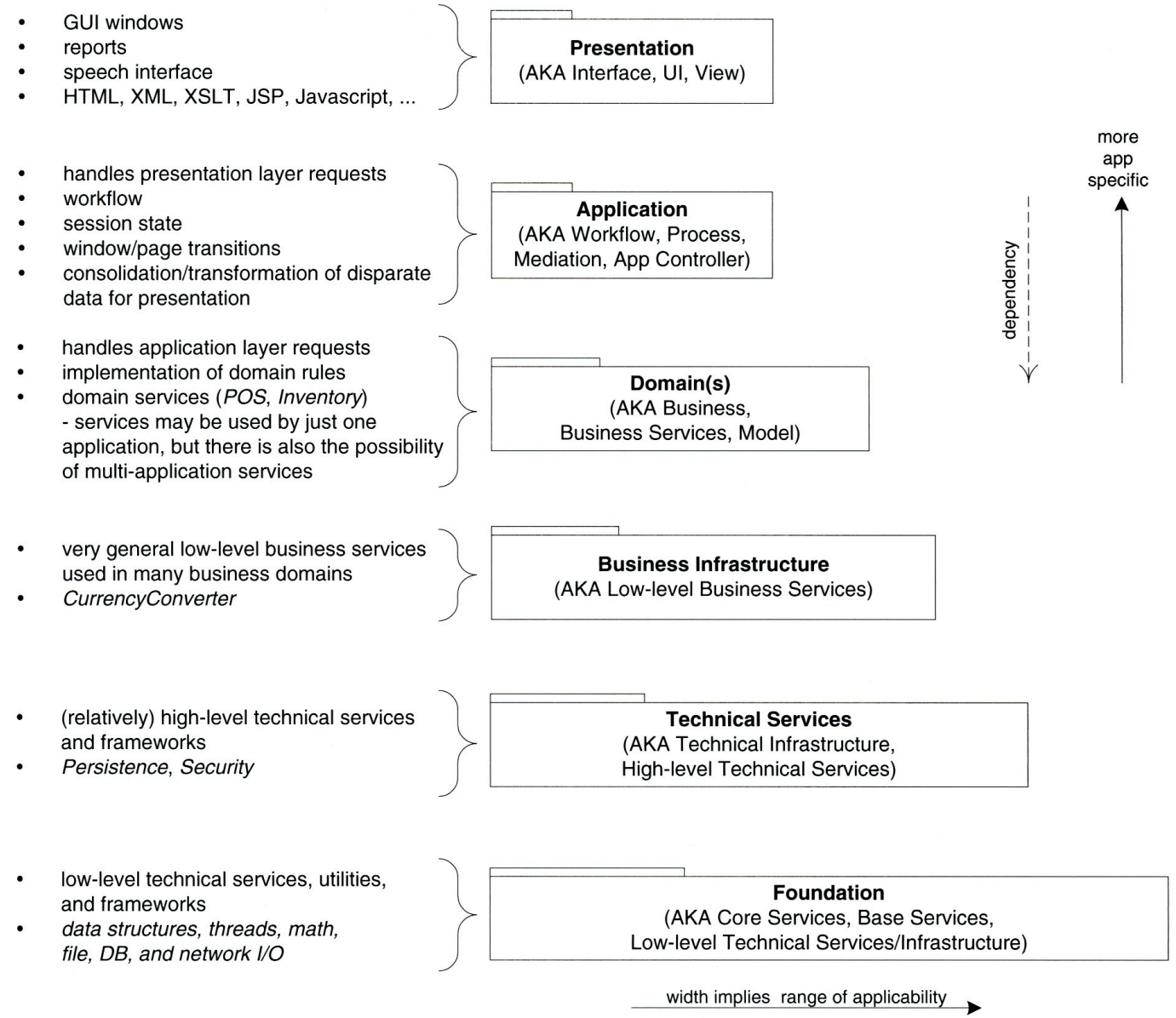
HSR Software Engineering 1 Vorlesung

Software Architektur: Schichten und Datenmodell

Daniel Keller

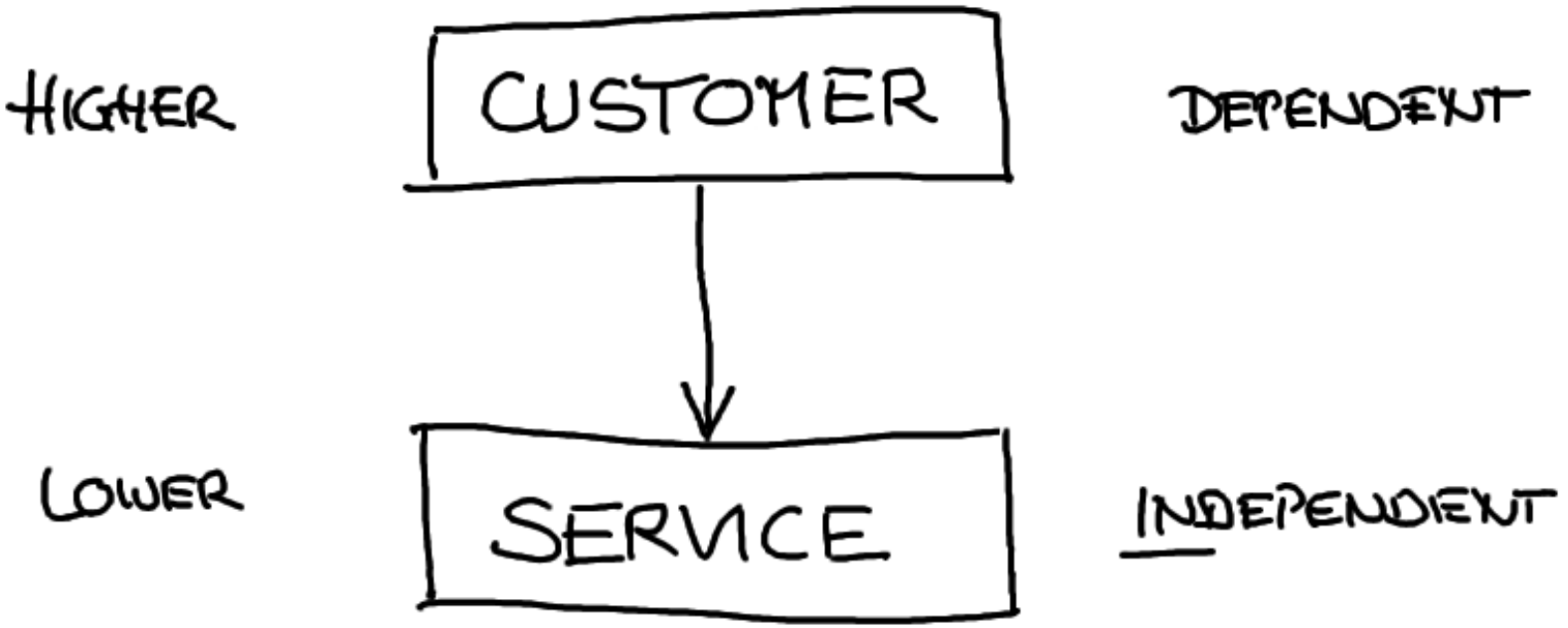
Herbst 2013

Schichten



Klassisches Schichtenmodell (Larman)

Oben und unten

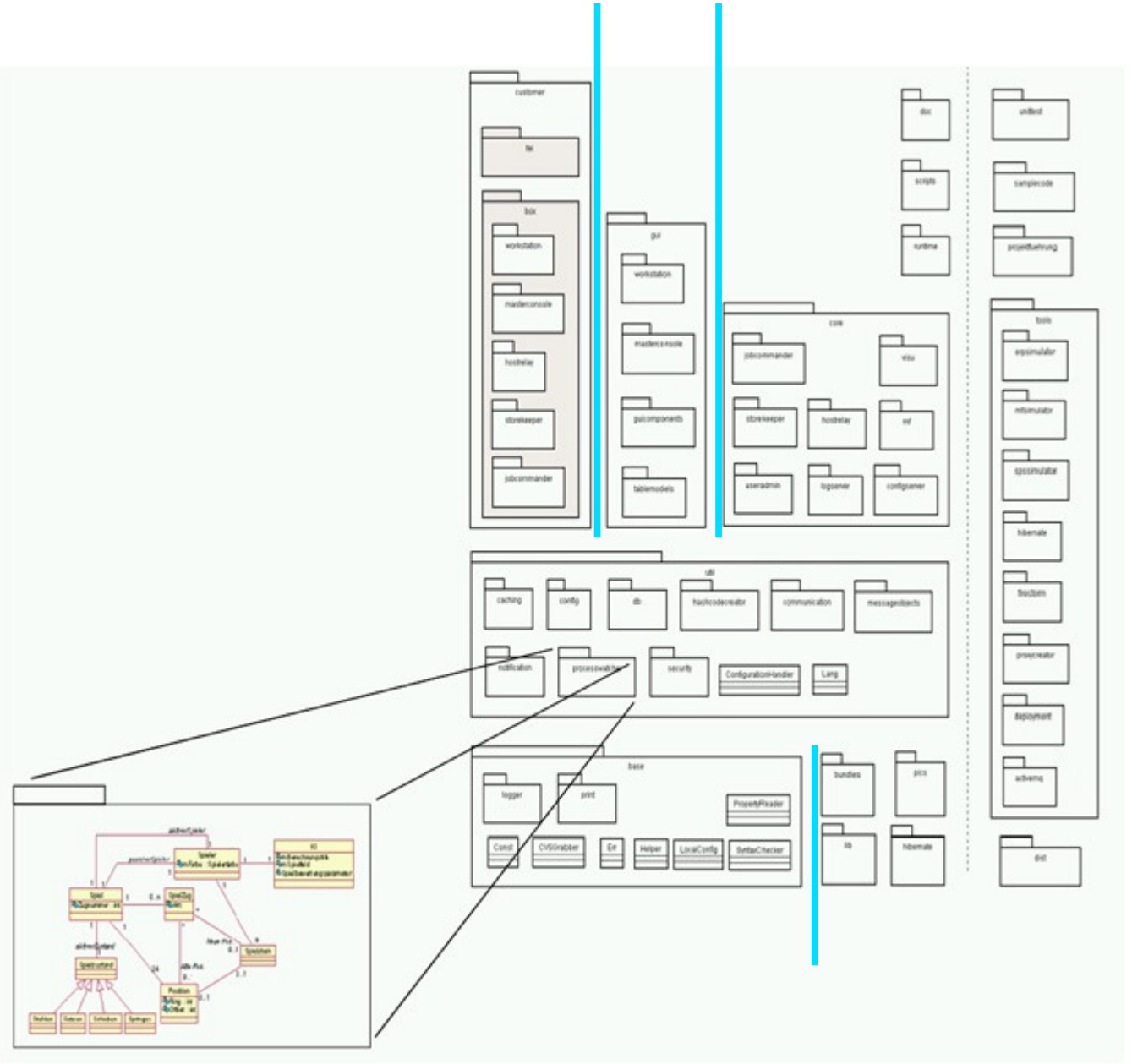


Schichten sind asymmetrisch!

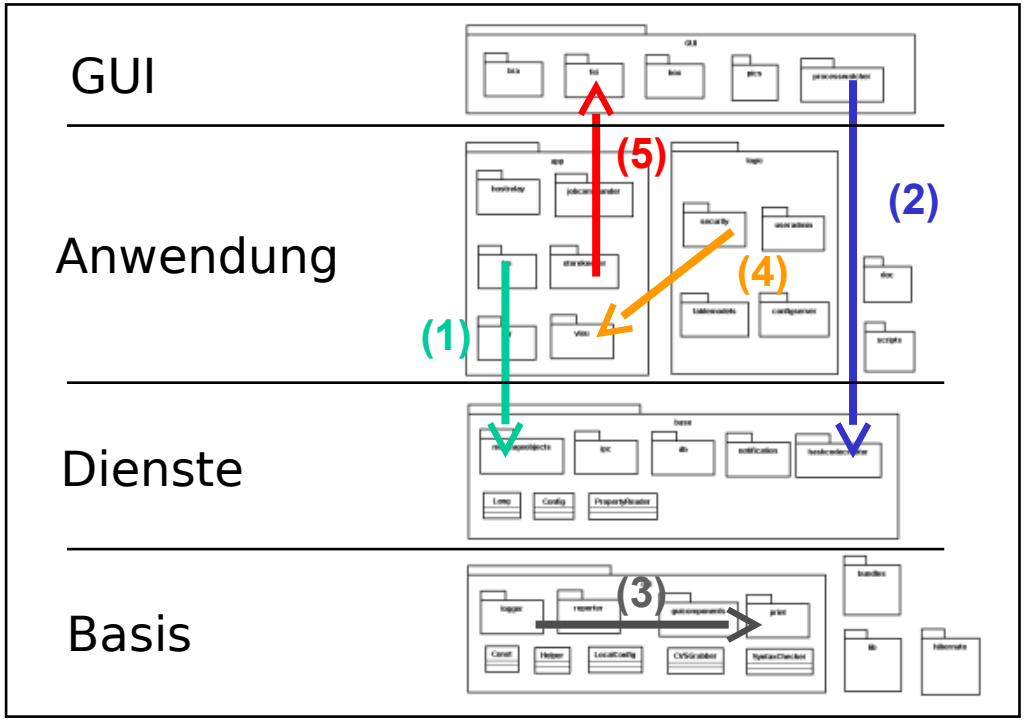
Abhängigkeit beachten.

Partitionen

Vertikale Unterteilung, d.h. gleiche Höhe der Pakete/Klassen, aber (möglichst) unabhängige Zuständigkeiten



Regeln für Abhängigkeiten



Aufrufe:

(1) … von oben auf die nächste darunterliegende Schicht sind immer OK

(2) … nach unten, die *eine* Schicht überhüpfen sind oft auch OK

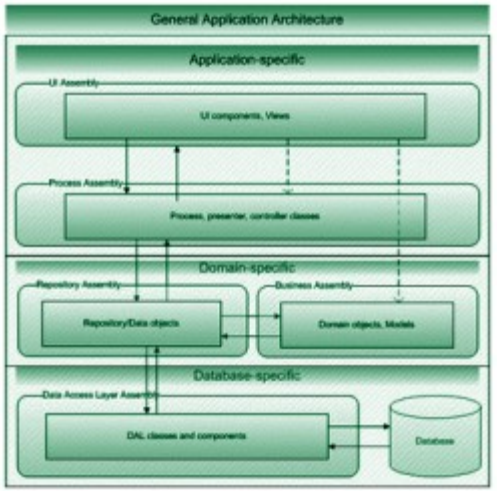
(3) … innerhalb einer Schicht und Partition sind OK, sollten aber minimiert werden

(4) … in einer Schicht quer über Partitionen sollten dringend vermieden werden

(5) … NIE von unten nach oben, ausser callbacks (z.B. Observer pattern)

Layers & Tiers

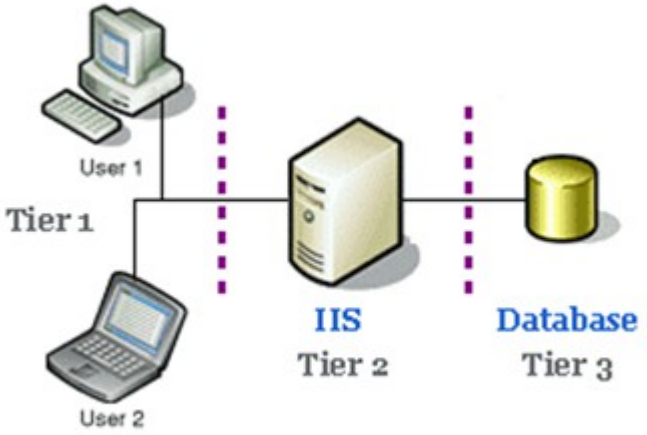
Schichten (Layers, horizontal)



„Wie der Code strukturiert ist"
hierarchische Abhängigkeiten,
klar, wer den Takt angibt

Tiers (vertikal)

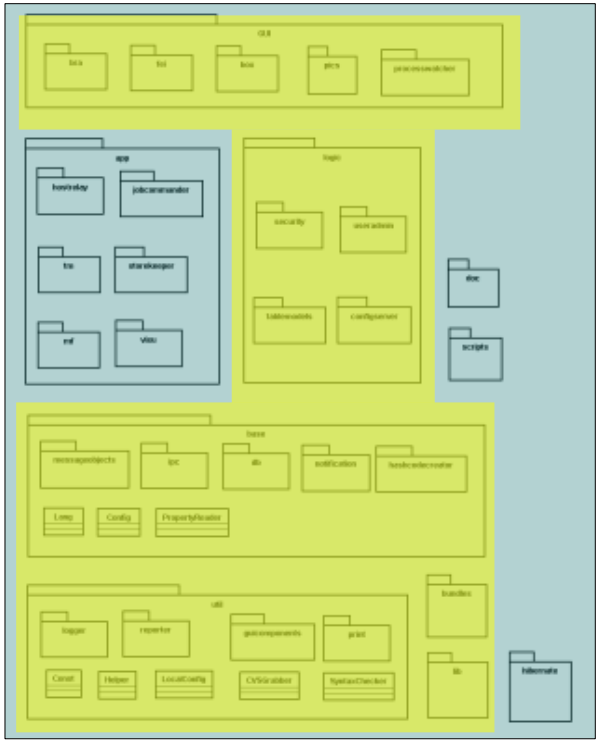
Bilder: davidhayden.com



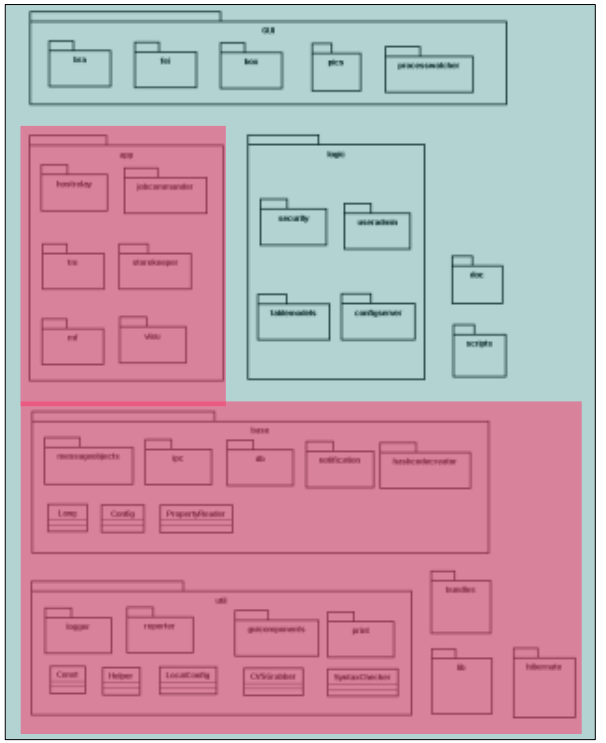
„Wo welcher Teil läuft"
i.d.R. gleichberechtigte Partner,
kein diktierter Takt

In einem System werden n Schichten üblicherweise auf n-x Tiers abgebildet (selten n+x). Beides sind Ansichten derselben Architektur: Klassen/Schichtendiagramm und Deployment diagramm.

Wo welcher Code läuft



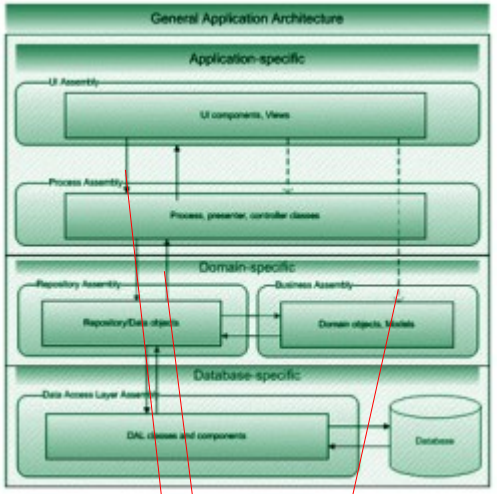
Clients



Server

Geteilter Code - als typisches Beispiel: Libraries - kann auf mehreren Tiers (=Maschinen) laufen

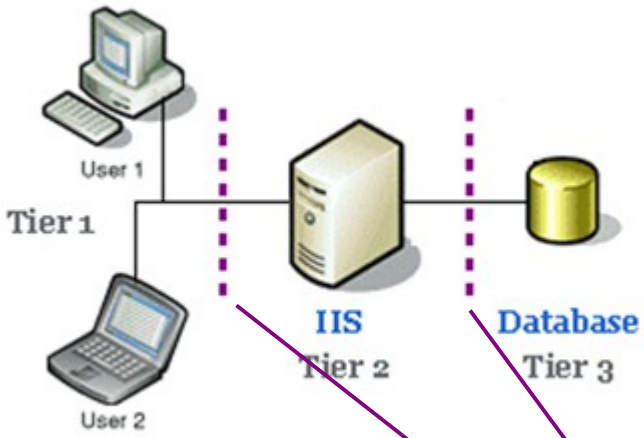
Kommunikation in Layers & Tiers



Layers

Typischerweise:
Aufrufe zwischen Schichten sind synchron

Tiers



Interfaces zwischen Tiers sind asynchron, d.h. potentiell langsamer und „teurer".

See: "Avoid Chatty Interfaces Between Tiers" Tiers should be minimized - impact on performance, scalability, security, fault tolerance, and complexity. http://davidhayden.com/blog/dave/archive/2005/07/22/2401.aspx

Entwurfs-Richtlinien

Ziele (im Grossen wie im Kleinen):

- Verbesserung der Verständlichkeit
- Verminderung der Komplexität
- Verminderung der Abhängigkeiten

Schwierigkeiten besonders im Grossen:

- Performance, Skalierbarkeit
- Verfügbarkeit
- Security
- Anpassungsfähigkeit (ständig änderndes Umfeld)
- ungewolltes Duplizieren

Kopplung und Kohäsion

Kopplung: Aufrufe von einer Klasse zur anderen (z.B. „feature envy“), von einem Package zum anderen

Kohäsion: Zusammenhalt innerhalb einer Klasse
weniger auch: Zusammenhalt innerhalb eines Packages
(„warum sind wir zusammen, was ist der gemeinsame Zweck?“)

Regeln:

- Kopplung (=Abhängigkeiten) minimieren
- Kohäsion soll gegeben sein, sonst gehören die Dinge nicht in eine Klasse, in ein Package.

Separation of Concerns

Klare Aufteilung der Zuständigkeiten:

- Immer nur ***ein*** Thema pro Klasse/Methode, das ergibt dann auch die erwünschte hohe Kohäsion.

Präzisierung:

- immer nur ein Thema pro Methode, alles andere ist Unsinn, keine Ausnahmen
- manchmal mehr als ein Thema pro Klasse (weil man öfter mal eine Klasse mit unterschiedlichen Fähigkeiten ausstatten will), aber aufpassen, Klasse nicht überladen.

Beispiele für Zuständigkeiten

Ausgabe (GUI, z.B. Warnungs-Popup, Datenreihe als Kurve darstellen, Ausgabe-Fenster öffnen...)

Formatierung der Ausgaben (z.B. länderspezifische Sprach-Strings, HTML/XML-Tags hinzufügen...)

GUI-Logik (welcher Knopf/Menu-Eintrag ist enabled/disabled, was ist der nächste Schritt/Dialog)

Benutzer-Eingaben ab Tastatur, Scanner oder GUI lesen (reine HW-Kapselung, Daten-Weiterleitung)

Eingabe-Validierung nach Lesen (z.B. Parsing, gültige Wertebereiche prüfen "muss eine Zahl zwischen 0.01 und 100 sein"...)

Programmlogik/Ablauflogik, Business Logic (Schritte, wann passiert was, wie oft wird wiederholt, wer hat welche Rechte, wann wird gelöscht...)

Algorithmen, Mathematische Berechnungen (z.B. Sortieren, Matrizenrechnung, Numerik, Trigonometrie...)

Kapselung von Hardware (z.B. Ansteuerung eines IR-Sensors)

Persistenz (dauerhafte Speicherung von Werten, z.B. in DB, als XML-Datei, ini-Datei, serialisiertes Objekt...)

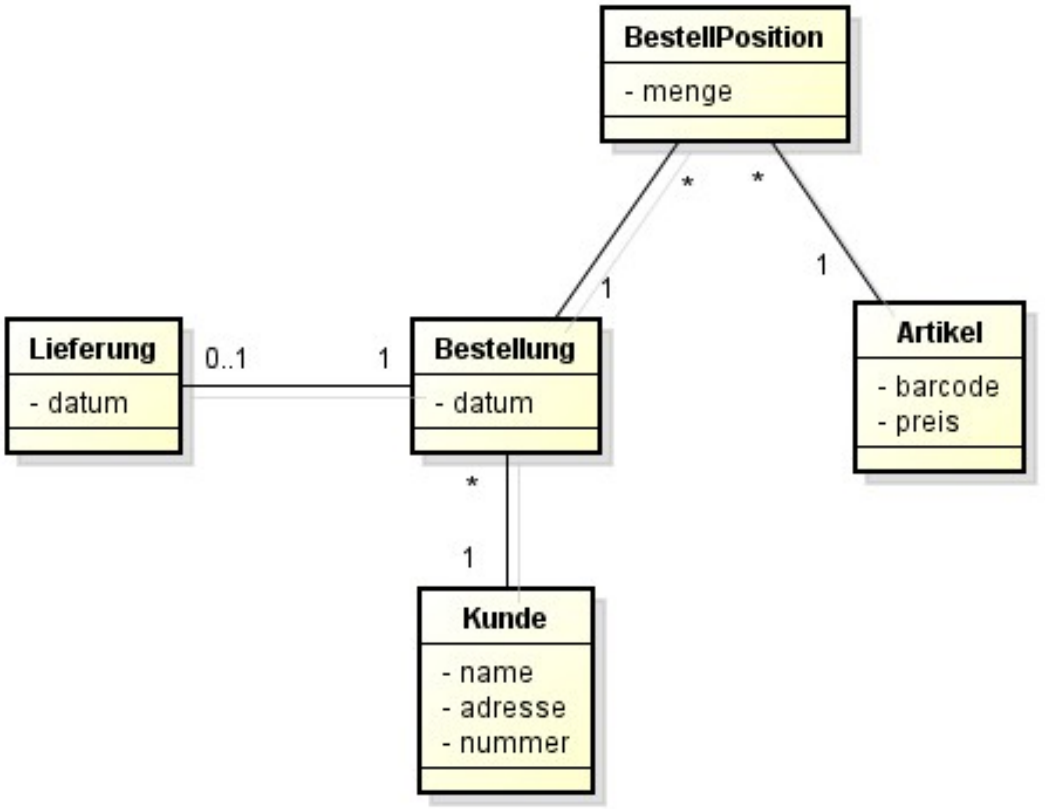
Grössenordnungen

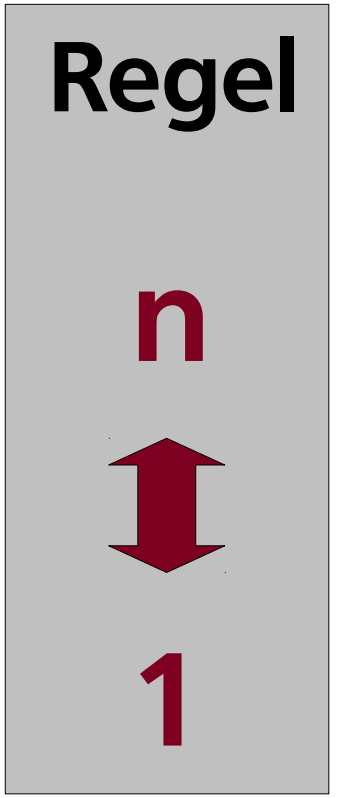
- Design-Diagramme (Übersichten) max. A3 (wg. Drucker und Beamer)
- Code-Zeilen: max. 120 Zeichen lang (u.a. wegen Beamern)
- Methoden (Funktionen): max. 1 Bildschirm, d.h. max. ca. 30 Zeilen
- Klassen: max. ca. 300 Zeilen
- Packages: keine Angabe, da abhängig von Funktion/Zuständigkeit
- Schichten: max. 7
- Tiers: max. 4 (Performance, Security)

(klar: immer mit Ausnahmen)

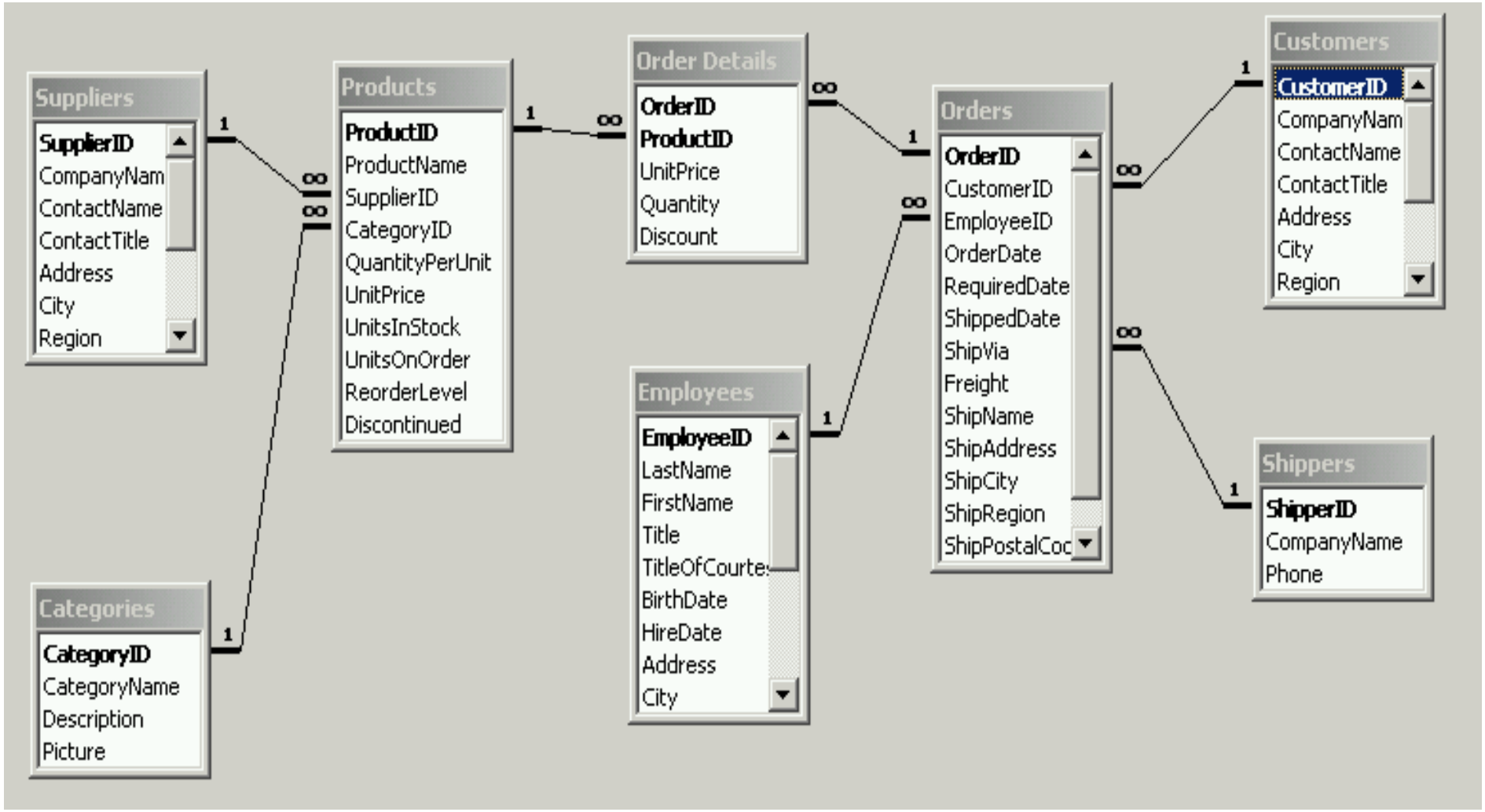
Regeln zur Anordnung

1. Eins-zu-n Beziehungen: *eins unten, n darüber*
2. Eins-zu-eins Beziehungen: *auf gleicher Höhe*
3. n-zu-m Beziehungen: *auf gleicher Höhe, immer mit Klasse darüber*
4. Ableitungen: *Basisklasse unten, abgeleitete darüber*



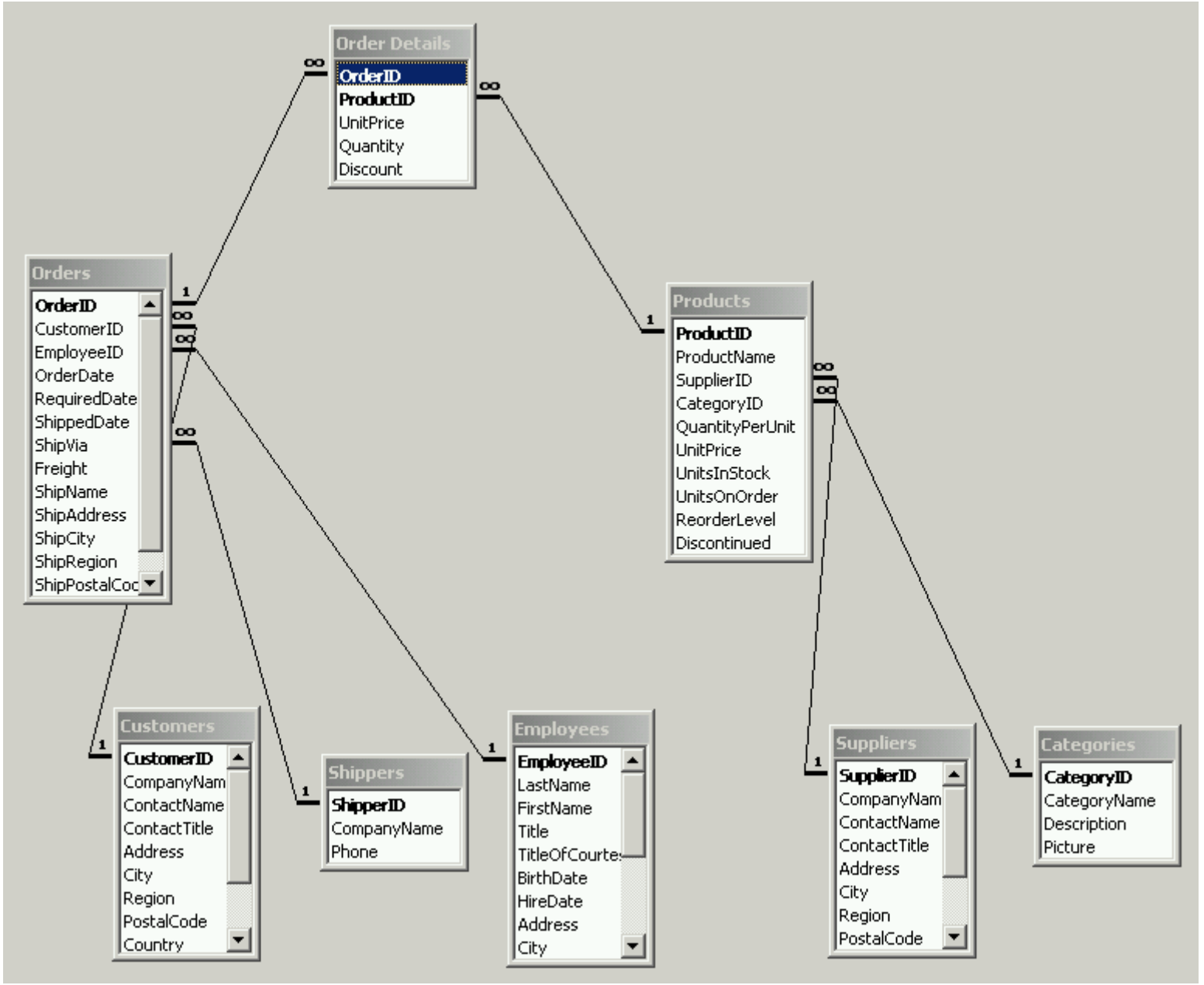


MS Access "Northwind" Beispiel (ungeordnet)

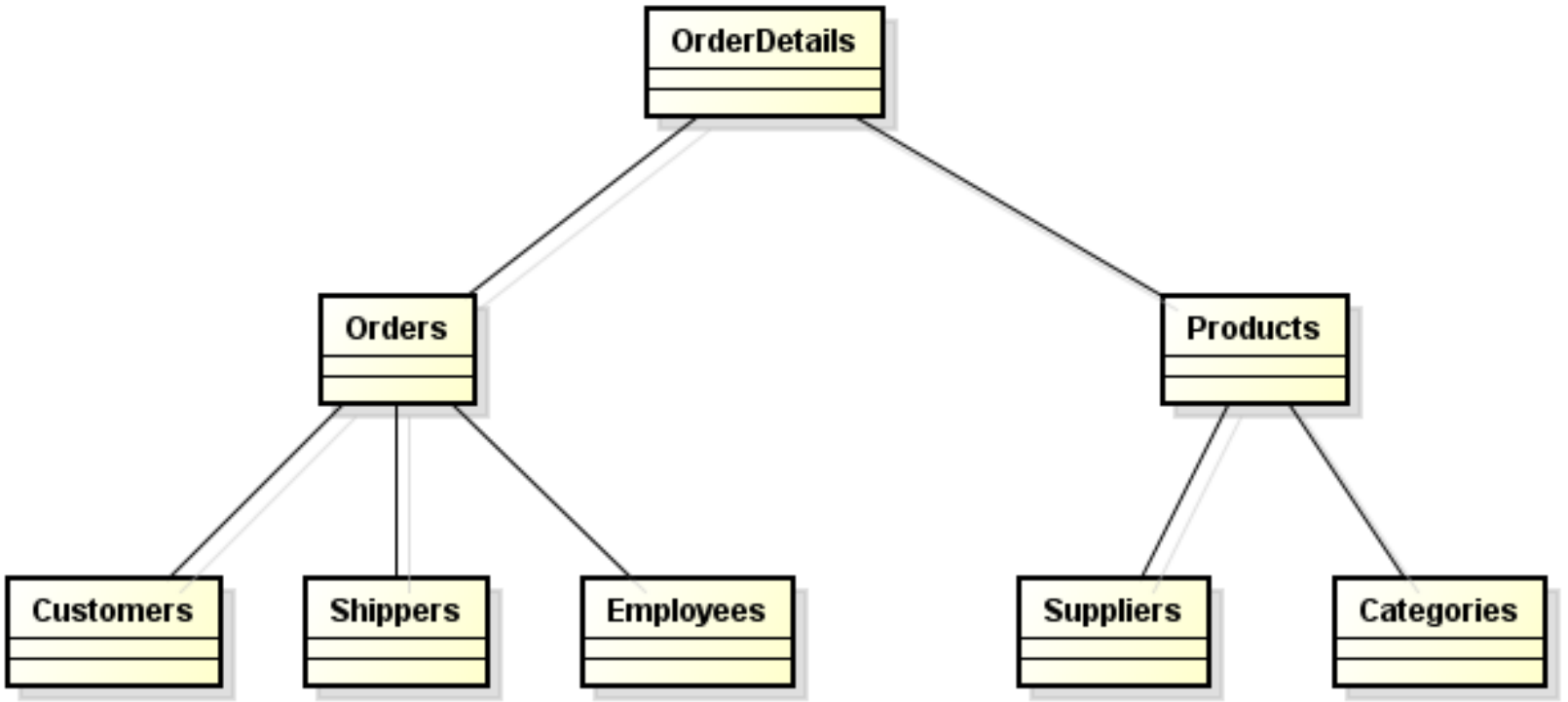


Jede 1:n Beziehung ist eine "Primärschlüssel - Fremdschlüssel" Beziehung

"Northwind" Beispiel geordnet

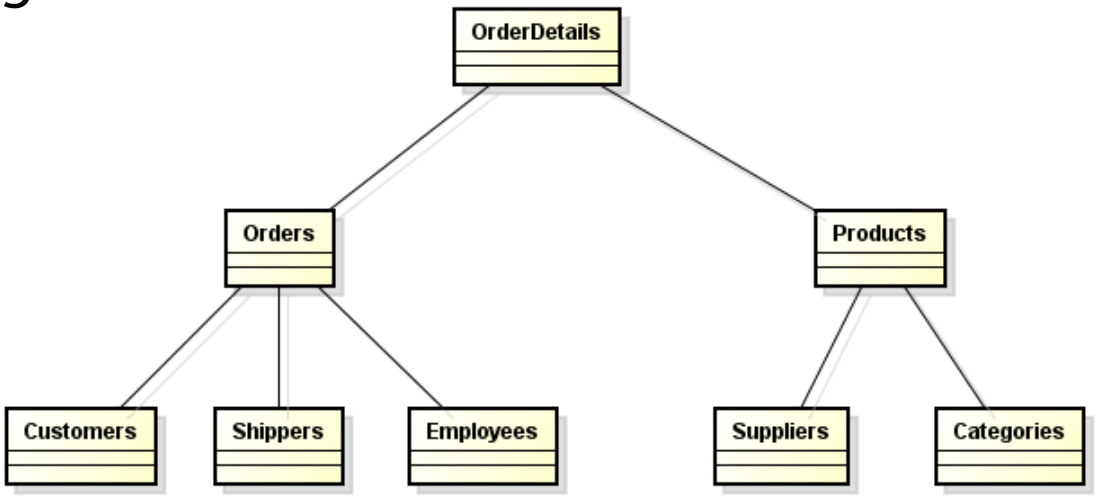


Gründe für's Ordnen



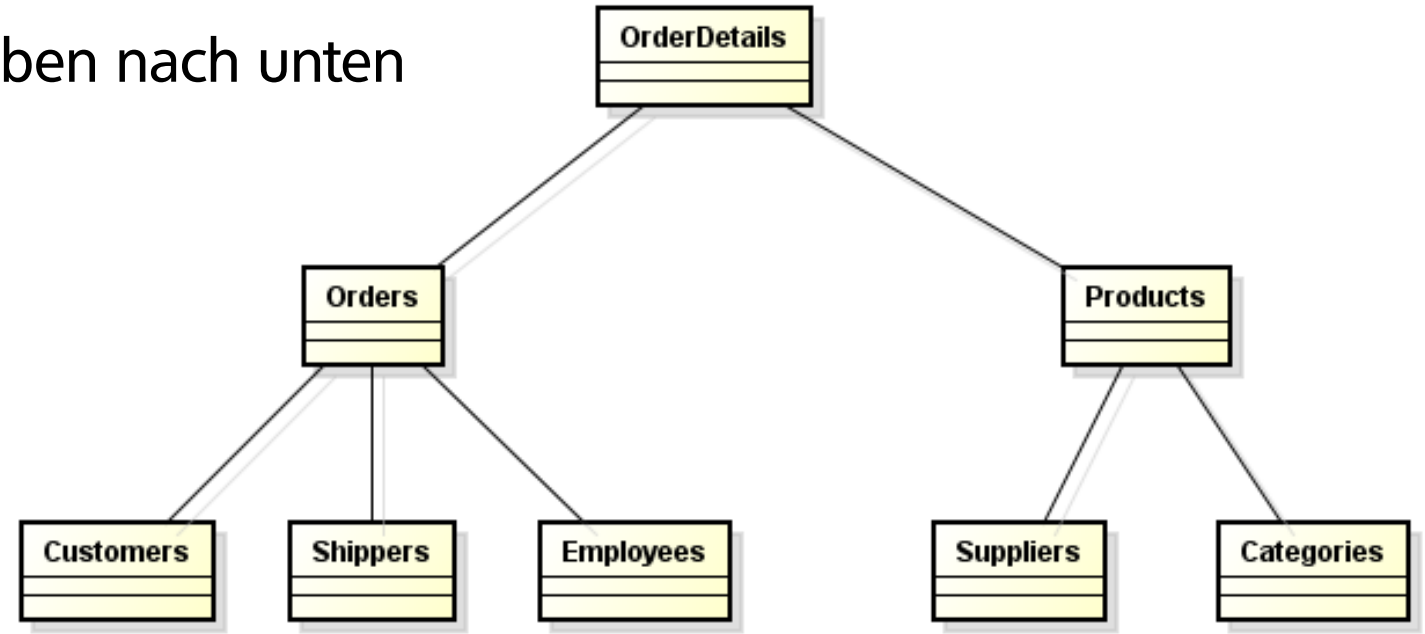
1. Mengen und Volatilität

- Die Klassen zuunterst ändern sich nur selten und leben lang *(Stammdaten)*
- Klassen zuunterst können einfach angefügt werden (kaum Randbedingungen), werden selten gelöscht (Abhängigkeiten)
- Klassen zuoberst sind zahlreich und eher kurzlebig *(Bewegungsdaten).* Sie werden häufig zugefügt, manchmal auch wieder schnell gelöscht. Aufpassen wegen Mengen (performance, table space) und insert/delete/archiving Kosten.



2. Füll- & Löschreihenfolge in DB

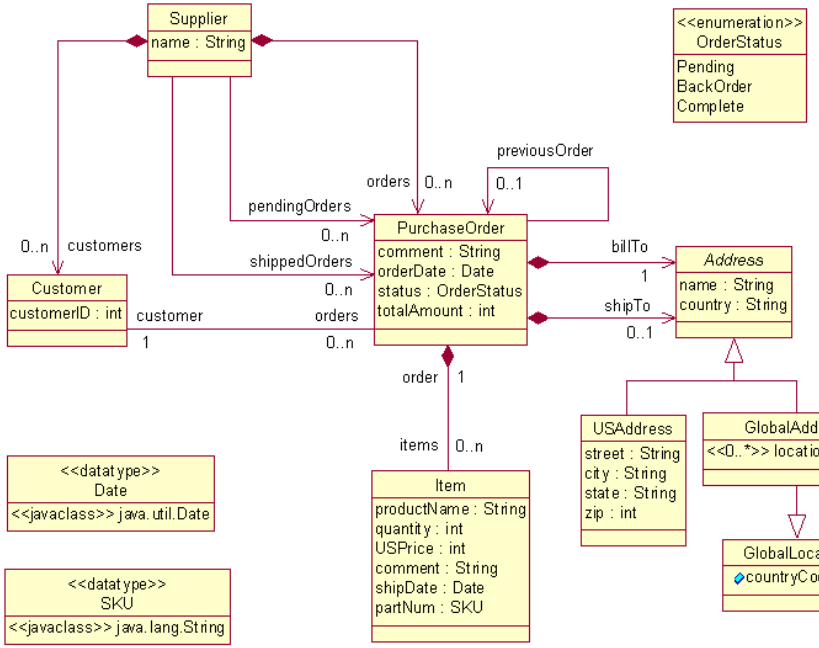
- INSERT INTO ... Anweisungen wenn man z.B. Testdaten generiert: die Reihenfolge ist kritisch, sonst kriegt man "unresolved foreign key references".
- Füllen: von unten nach oben
- Löschen: von oben nach unten



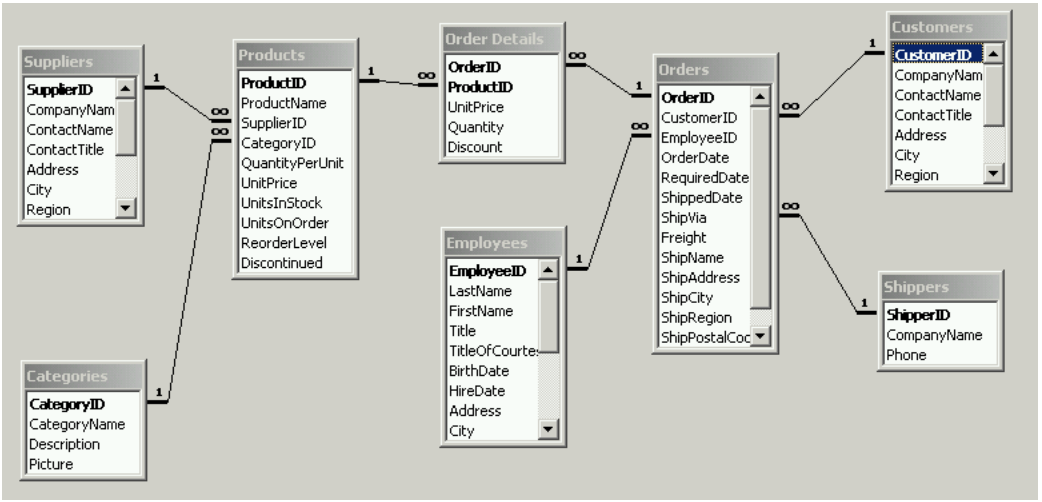
3. Compare, Diff, Merge

Schwierige Frage: was ist ein "Diff" von zwei Diagrammen?

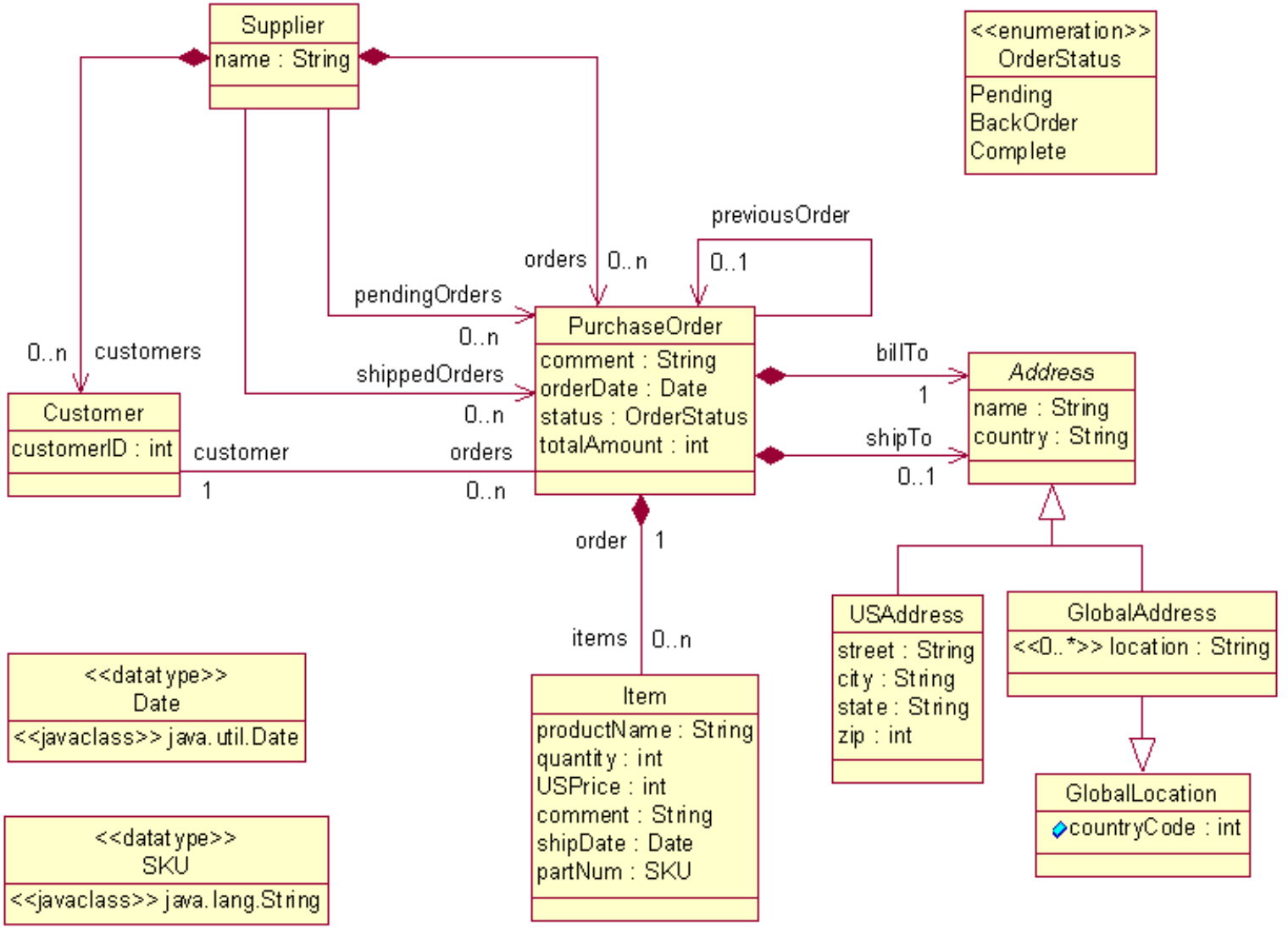
Was mit Texten ganz einfach funktioniert, ist mit Diagrammen fast unmöglich.



F. Budinsky, D. Steinberg, E. Merks, R. Ellersick, and T. J. Grose.
Eclipse Modeling Framework. Pearson Education, Inc., Boston, MA, 2003.

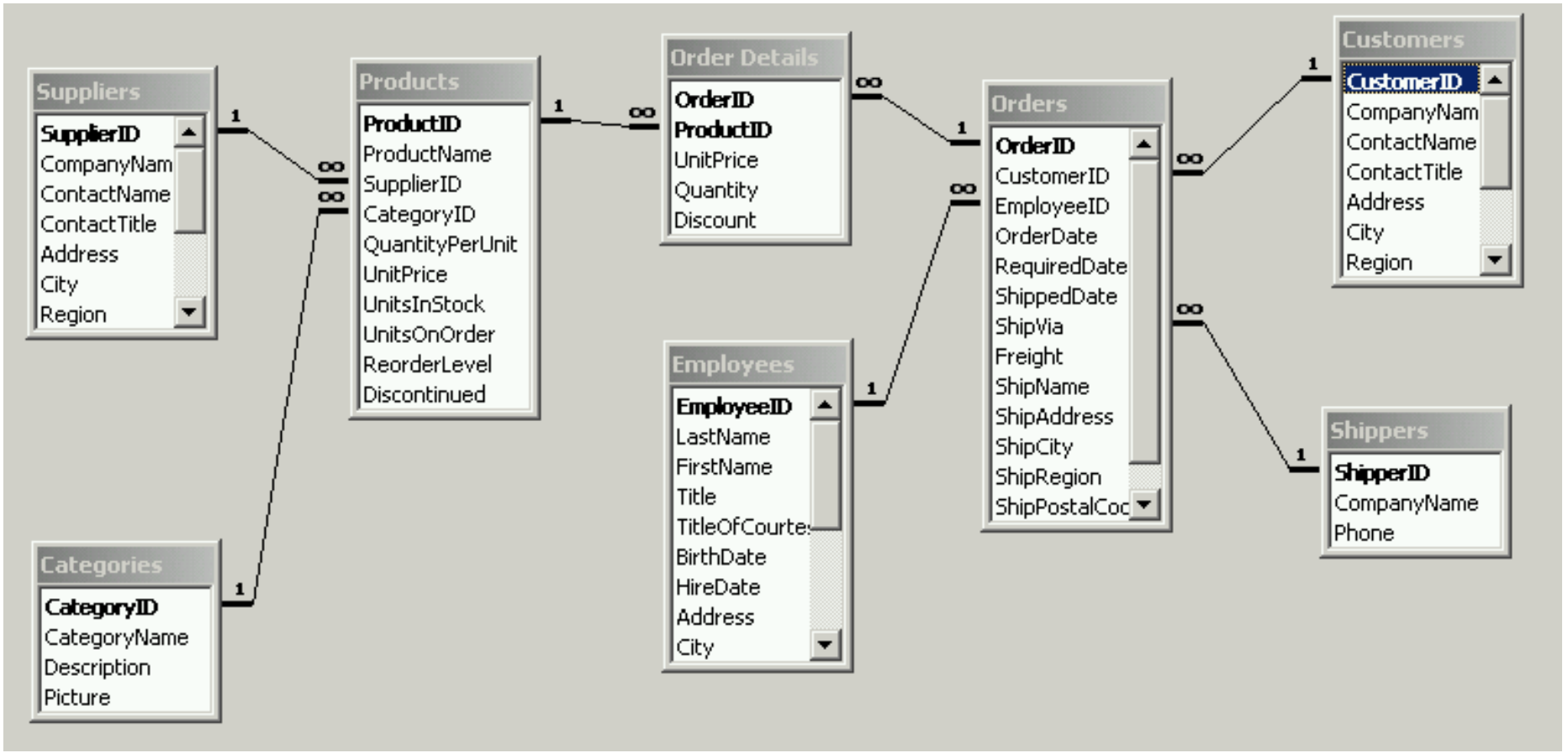


Beispiel für ein Domain Model

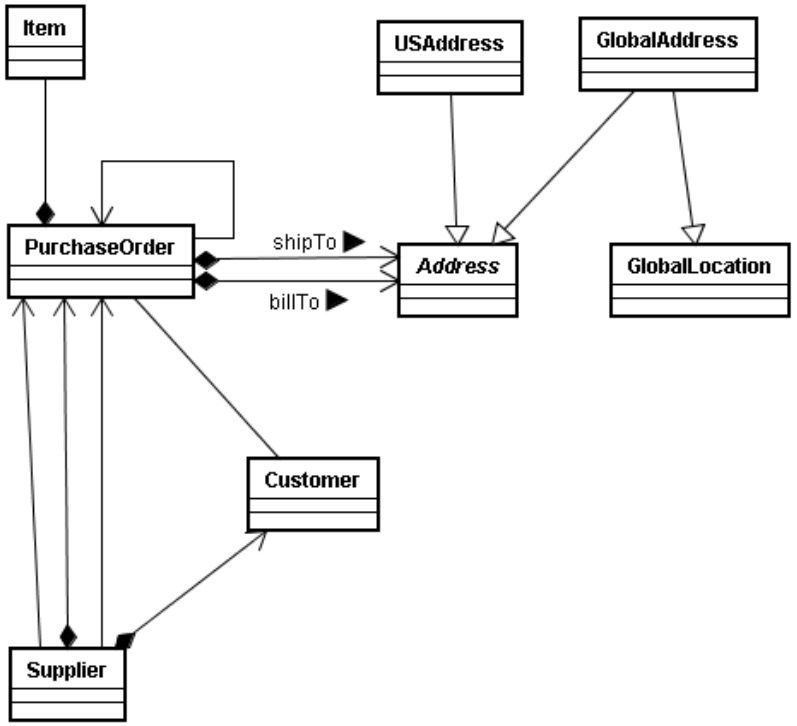


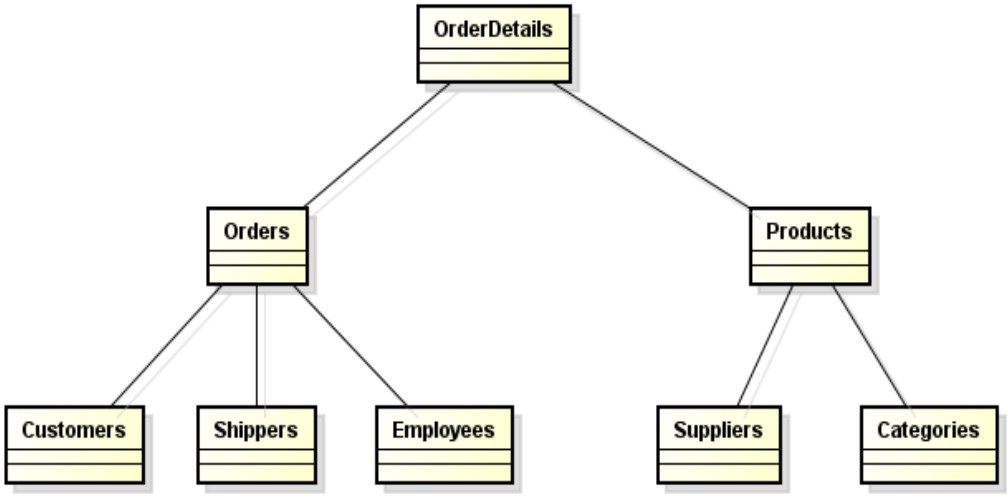
F. Budinsky, D. Steinberg, E. Merks, R. Ellersick, and T. J. Grose. *Eclipse Modeling Framework*. Pearson Education, Inc., Boston, MA, 2003.

Vergleich zu vorigem Diagramm?



Entitäten sortieren

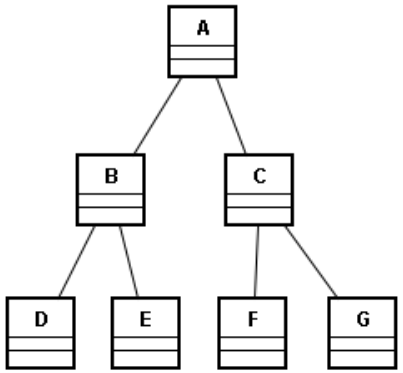




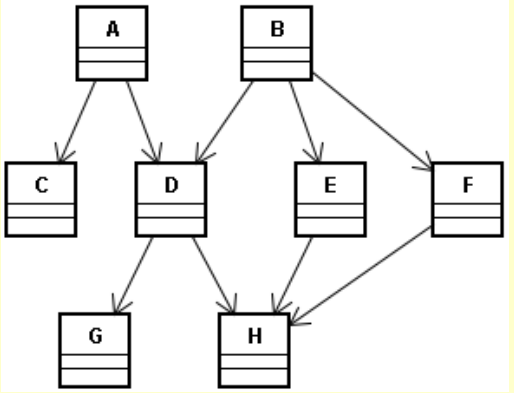
Jetzt kann man vergleichen oder zusammenführen (diff/merge)

4. Loop Detection

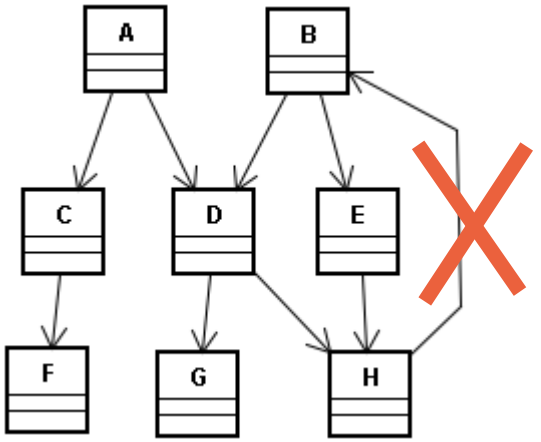
Ein Datenmodell sollte immer in der Form eines gerichteten, azyklischen Graphen vorliegen (Directed Acyclic Graph, DAG, so ungefähr ein Baum mit Querverweisen, aber ohne Schleifen)



Tree (binary)



DAG



Graph with loop

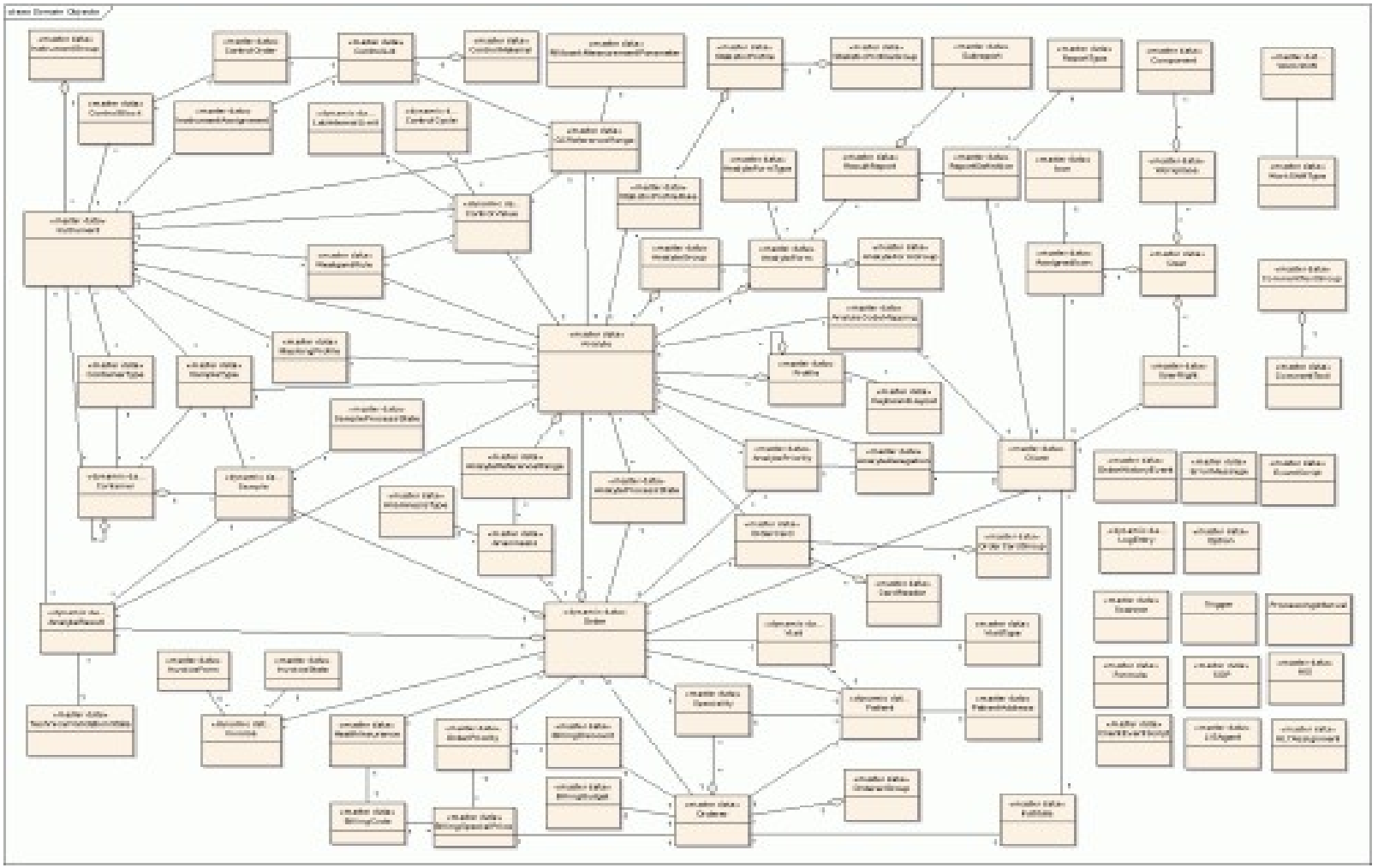
Warum keine Schleifen?

Wäre ein riesiger Fehltritt in einem Datenmodell:

- möglicherweise unendlich laufende oder unzulässige SQL statements (joins, inserts)
- unerwünschte/unvorhersehbare Trigger-Abhängigkeiten (eventuell endlos feuernde Kaskade)
- verringerte Chancen, das Datenmodell zu partitionieren: Kopplung zu eng.
- Widerspricht den Absichten einer Schichtung

Bemerkung: UML Design Class Diagrams können ausnahmsweise Schleifen enthalten.

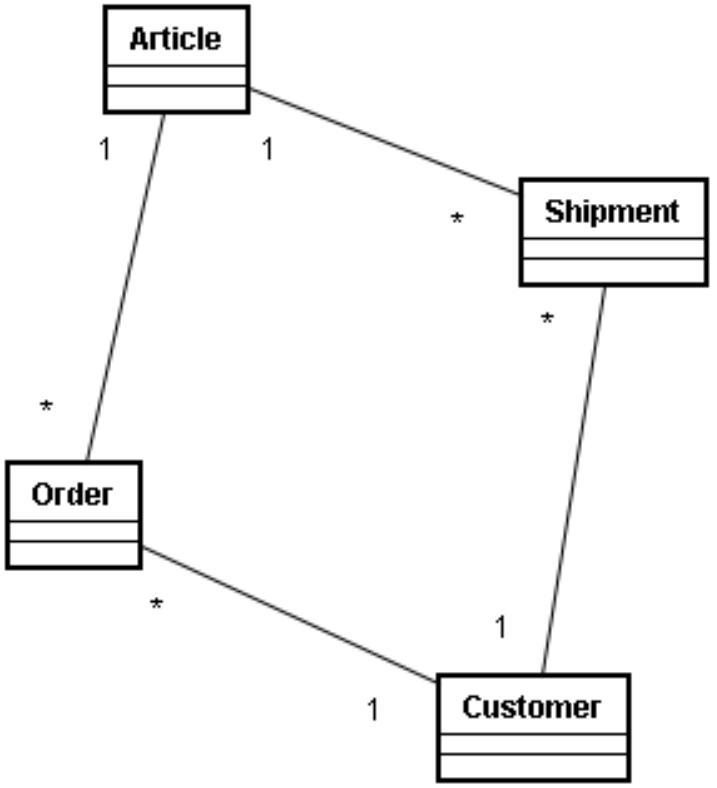
Finden Sie die Schleifen hier?



Echtes Beispiel eines data/domain model:
ca. 40 Programmierer und kein SW Architekt in den letzten 4 Jahren

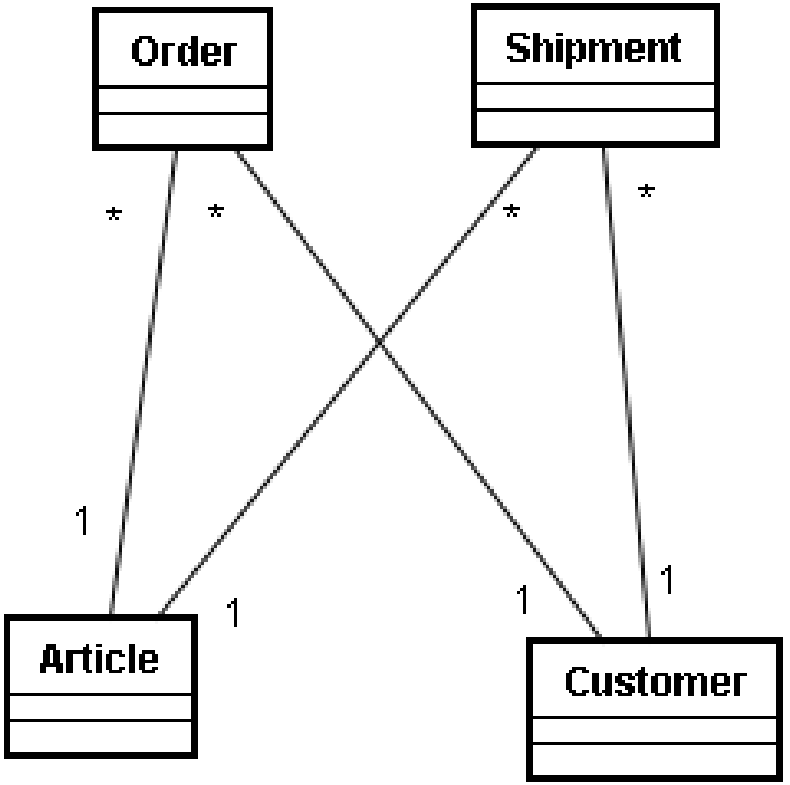
5. Multiple Path Detection

- Wann immer Sie ein Vieleck (meist Rhombus) in Ihrem Datenmodell finden, dann ist es Zeit, das genauer anzuschauen: es könnte OK sein, es könnte extra Vorsicht verlangen (extra constraints) damit es funktioniert, oder es könnte schlicht falsch sein.
- Sie brauchen immer eine Erklärung für Mehrfach-Pfade.

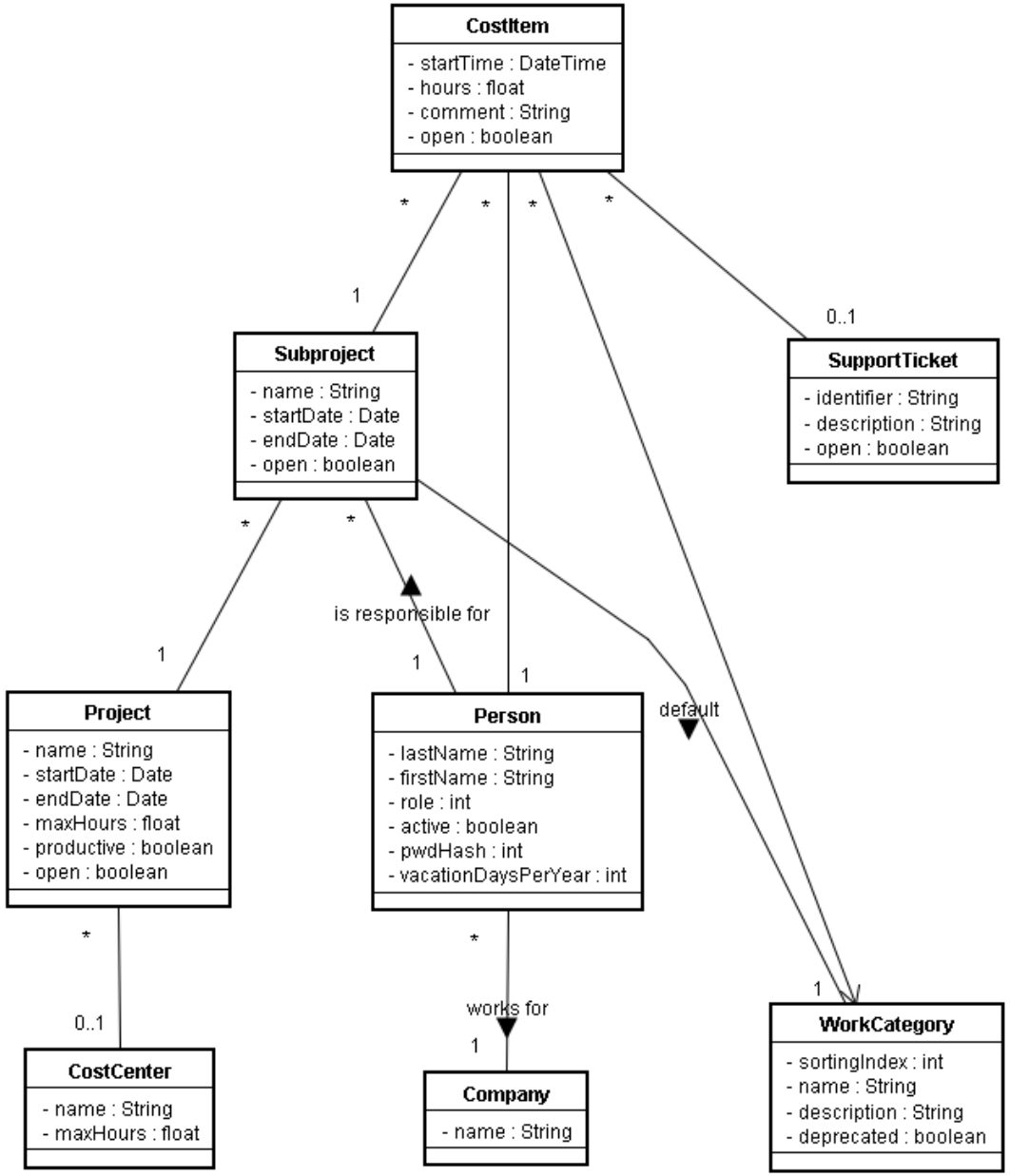


Typisch: "Doppelgabel"

- Wenn Sie das Datenmodell ordnen, dann sehen Sie sofort, ob das Rhomboid eine "Doppelgabel" ist (normalerweise OK) oder ob es immer noch ein Rhomboid ist, das genauer angeschaut werden muss.

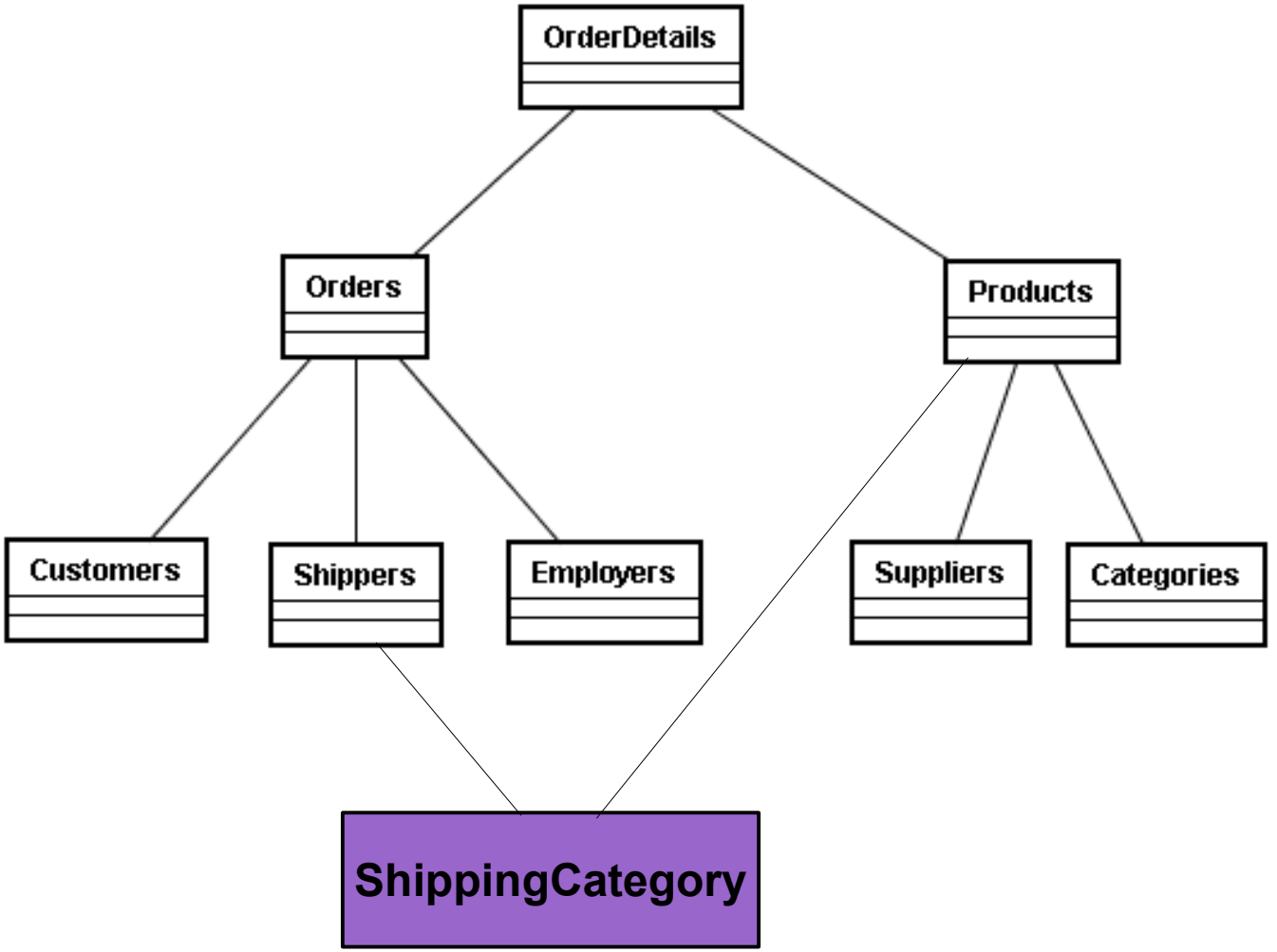


Beispiel: Time Tracking



- Die "default WorkCategory" ist fragwürdig.
- Die "person responsible" braucht eventuell noch mehr Programmierung: nicht alle Personen dürfen Time Sheets unterzeichnen; darf jemand für alle in der Gruppe unterzeichnen oder nur für die im gleichen (Sub-)Projekt?; Niemand soll das eigene Time Sheet unterzeichnen dürfen.

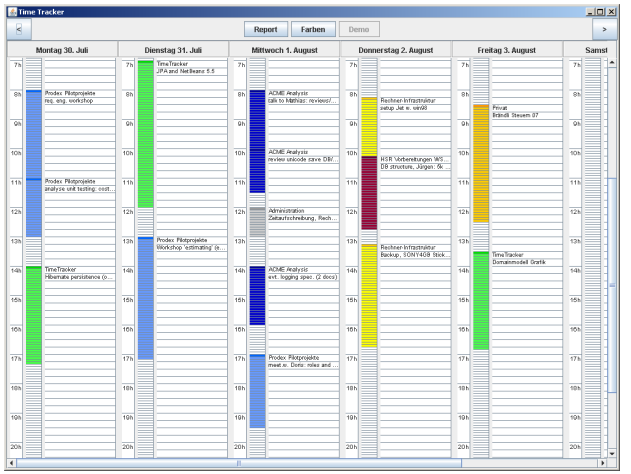
Example: Logistik



Mehrfachpfade können völlig OK sein (die neu zugefügte "Shipping Category": zerbrechliche Produkte gehen mit spezialisierten Spediteuren)

6. GUI Komplexität

Entitäten ganz oben brauchen meist sehr aufwendige GUIs: GUIs für erlechterte Eingaben (weil es häufig vorkommt), GUIs mit vielen Checks und Regeln (weil viel Geschäftslogik darunter ist).



Example screen "TimeTracker"

Operationen auf den Entitäten zuunterst sind praktisch immer ganz einfach "Create, Read, Update, Delete" (CRUD nach Larman). So ein GUI kann man mit Generatoren machen.

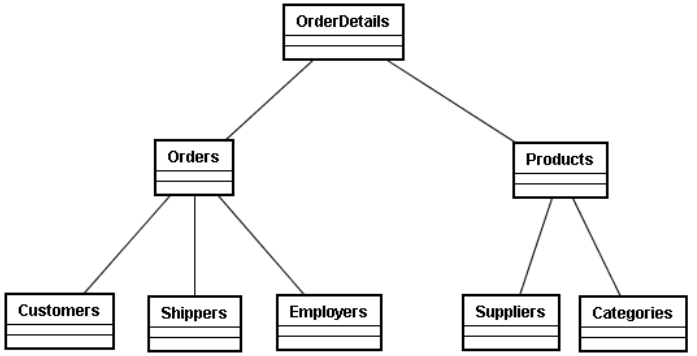
7. Layering, Partitioning

Manchmal will man die Datenspeicher-Klassen unterscheiden, z.B. die zuoberst nur „in-memory“, die in tieferen Schichten persistent (Beispiel Real-Time Messwert-Verarbeitung).

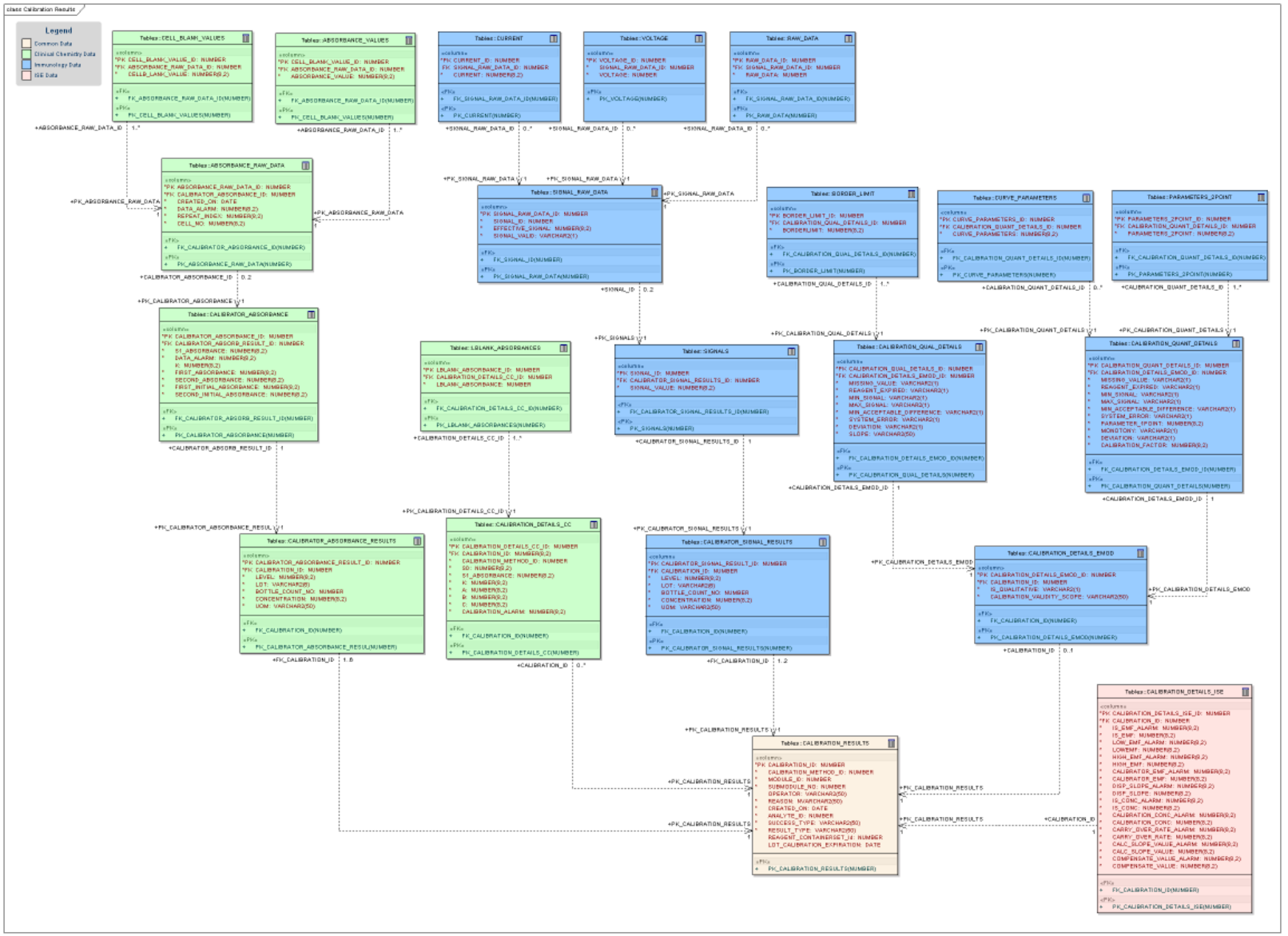
Manchmal ist es nützlich, den gesamten Datenbereich vertikal zu unterteilen, z.B. die Auftragsbearbeitung (SAP) von den physischen Warenhaltungs-Operationen (Lagerlogistik) zu trennen (wer darf jeweils die Dinge erstellen/modifizieren/löschen, mit Lesezugriff für alle)

In beiden Fällen möchte man mit einem Schnitt möglichst wenige Linien (Abhängigkeiten) schneiden (Baum ideal für vertikale Partitionierung, DAGs nicht mehr ganz so einfach, Schleifen können es verunmöglichen)

Ein perfekter Baum:



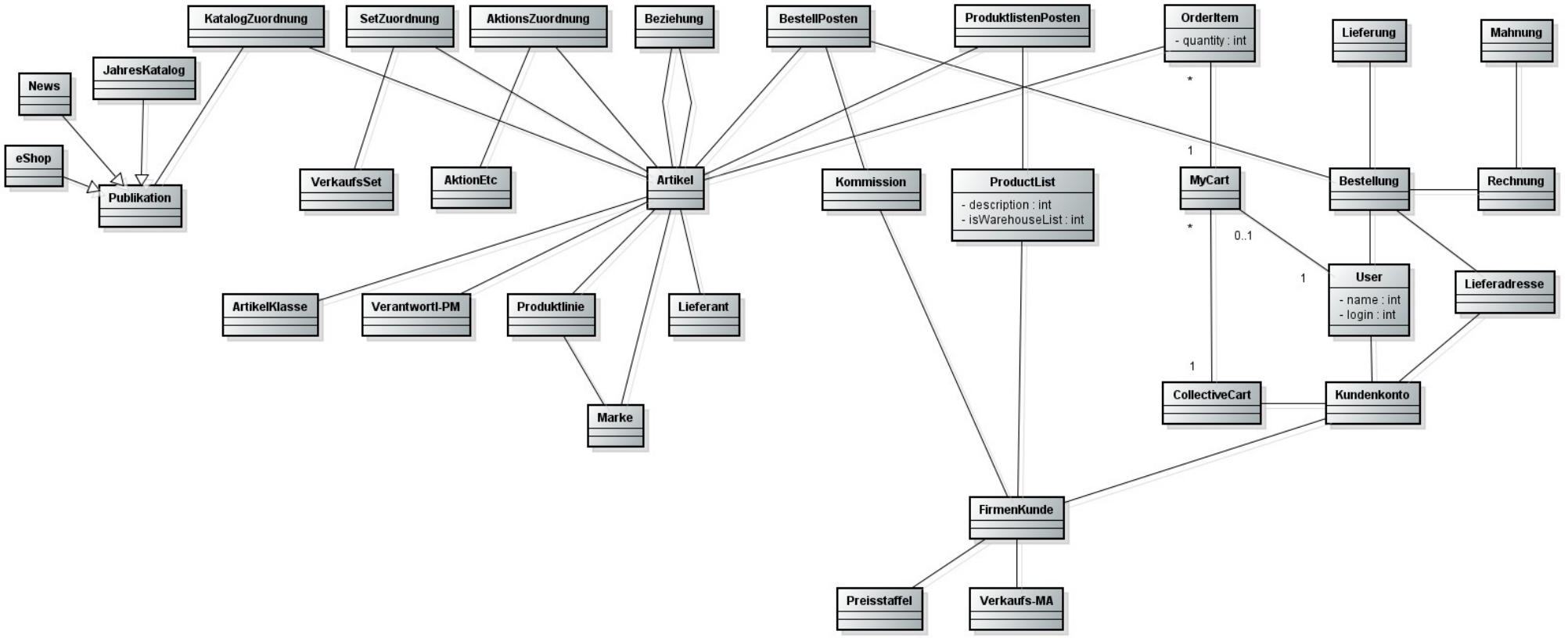
Sauberes Datenmodell



Echtes Datenmodell in der Form eines umgekehrten Baumes, partitioniert

Daten aufteilen

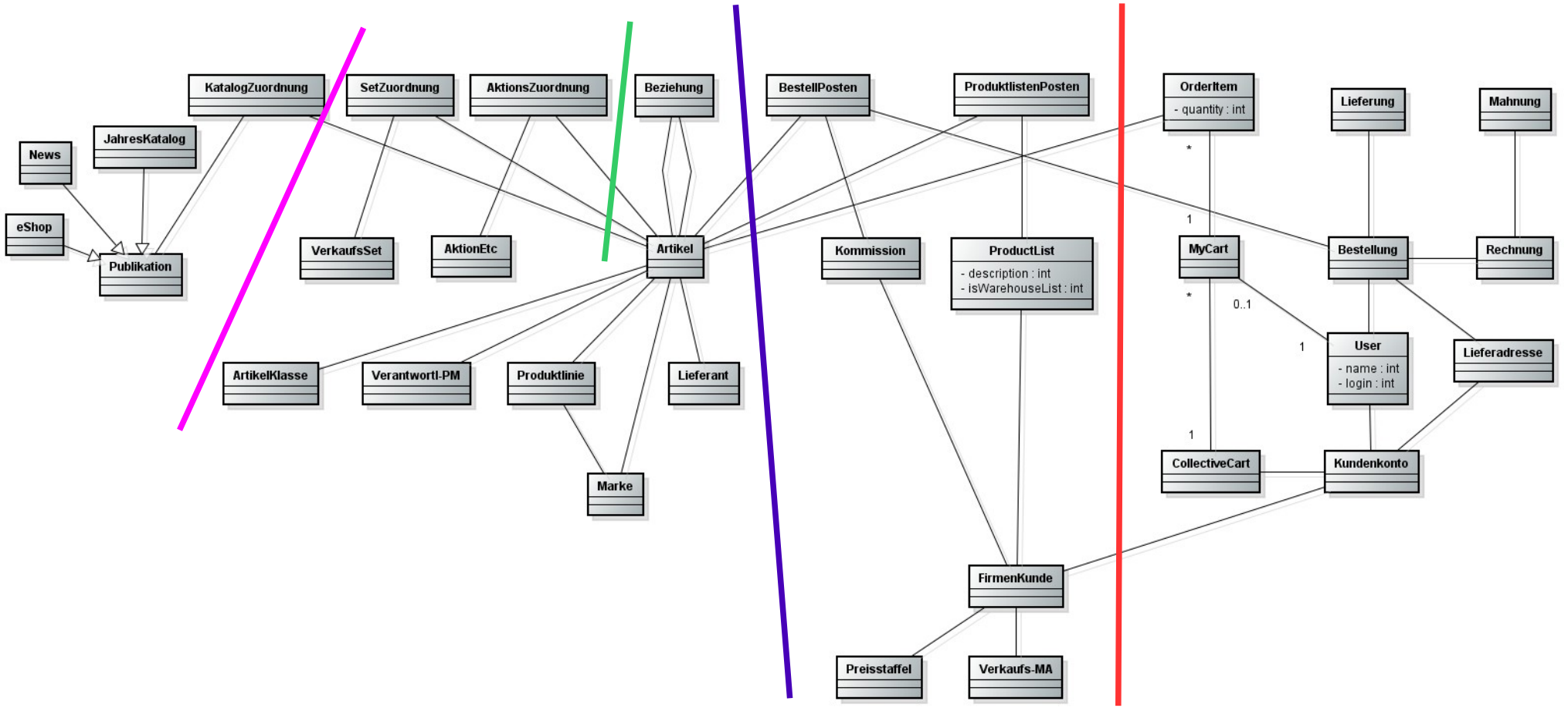
... und damit auch Verantwortlichkeiten.



Eine Schnittlinie sollte möglichst wenig Beziehungen durchschneiden.

Abhängigkeiten beim Schneiden

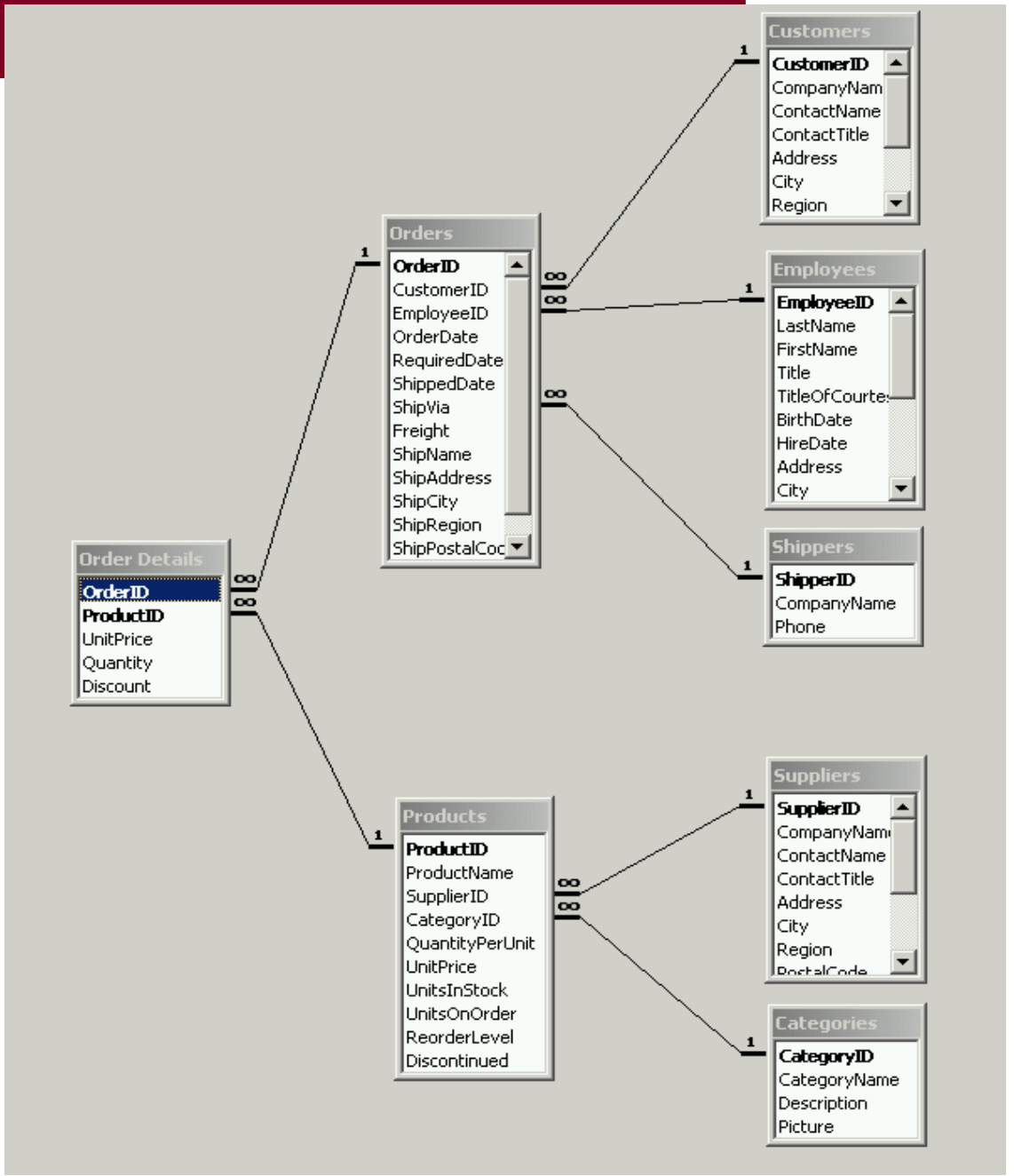
Eine Schnittlinie sollte nur Beziehungen in EINER Richtung durchschneiden.



Warum „oben-unten“ ordnen?

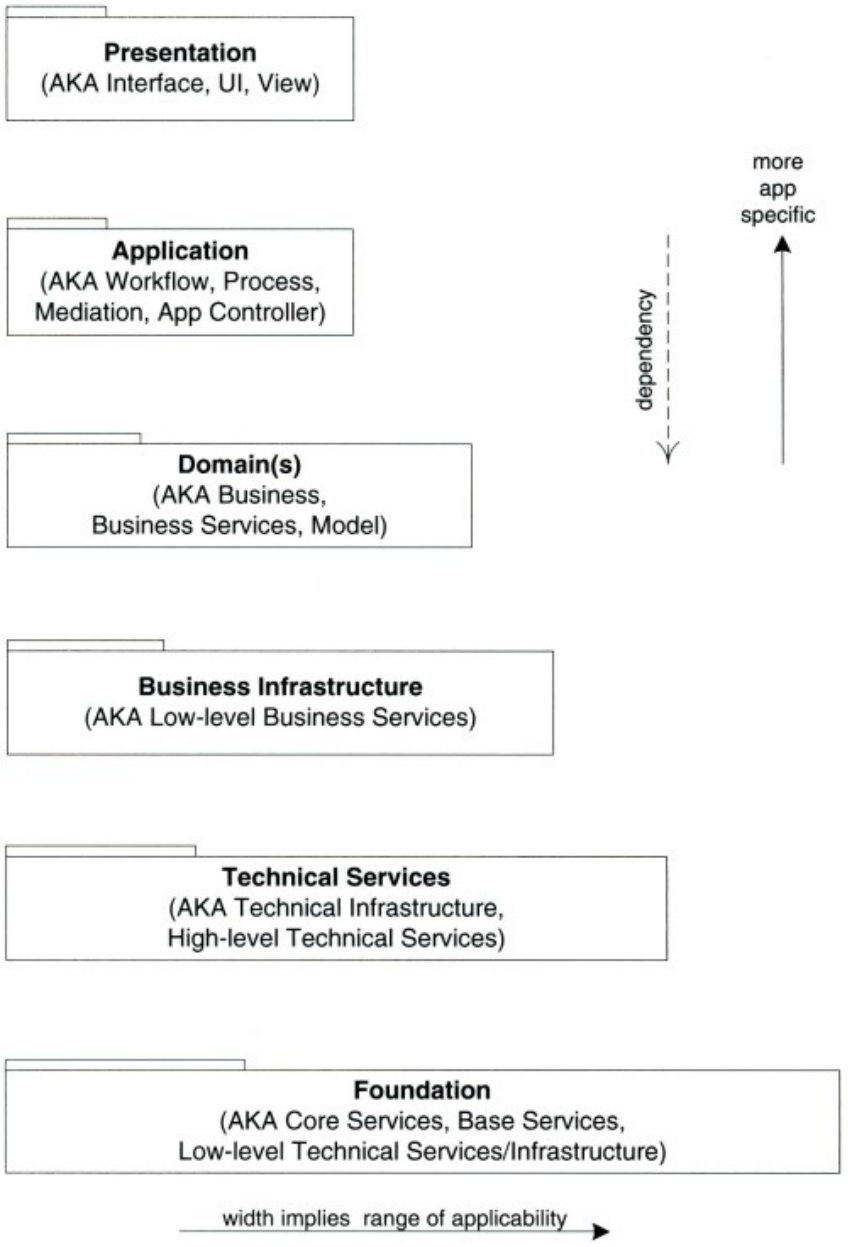
Man kann auch gut von rechts nach links ordnen.

Dies könnte besonders in Daten-zentrierten Systemen nützlich bzw. besser akzeptiert sein, z.B. ist ein LEFT JOIN so gut visualisierbar.



Traditionelle Schichtung

- Zuunterst sind die „low-level services“, darüber aufbauend immer komplexere Dienste.
- Analog setzt dazu die Objekt-orientierte Schichtung die Basis-Klassen zuunterst und die abhängigen (= darauf aufbauenden) Klassen darüber.
- Im Datenmodell sind die Stammdaten die "base classes“, die zuunterst liegen.
- In einer Objekt-orientierten Umgebung ist die „oben-unten“ Anordnung die naheliegende.

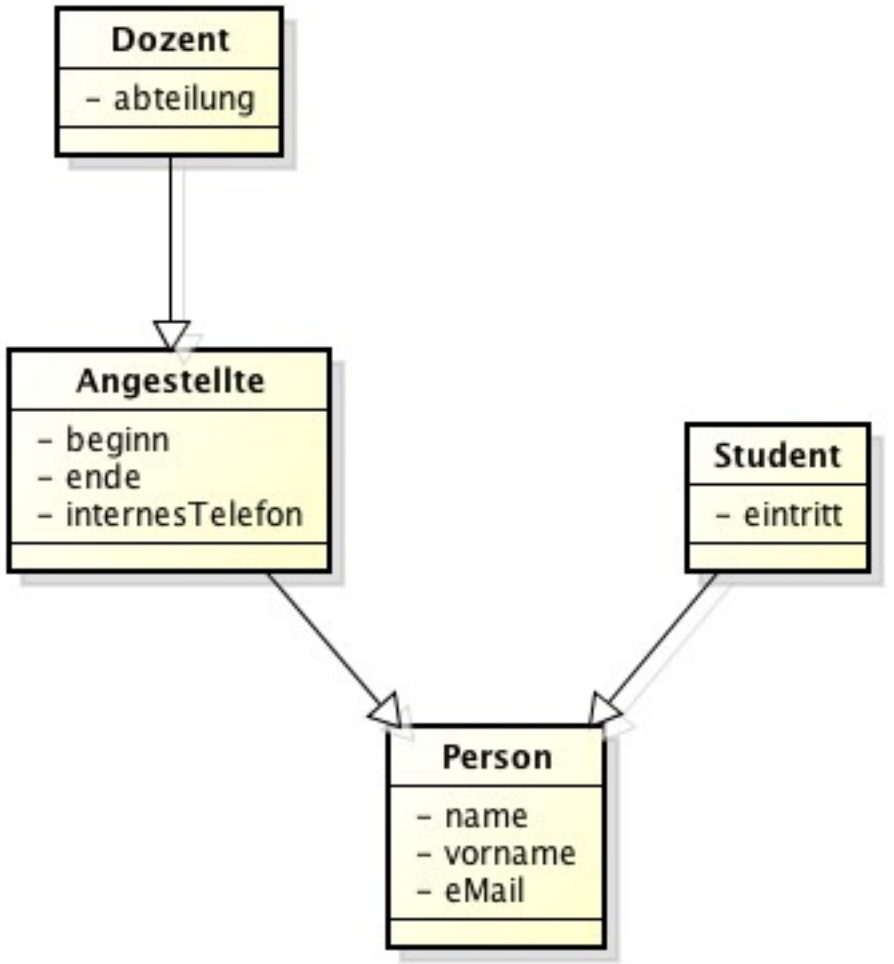


From: Craig Larman, Applying UML and Patterns

Vererbung: Basisklasse unten

Normalerweise sieht man einen Vererbungsbaum mit der Wurzel (!) zuoberst und die abgeleiteten Klassen hängen darunter.

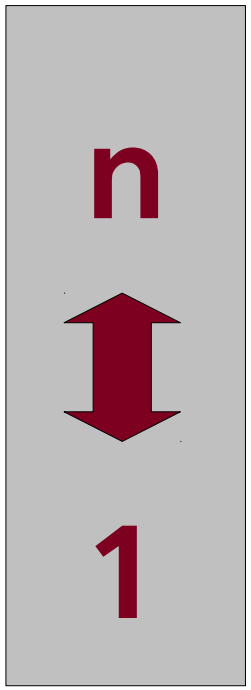
Logischerweise sollten aber in der objektorientierten Programmierung die Basisklassen unten sein, und die abgeleiteten Klassen darüber



From: Craig Larman, Applying UML and Patterns

Zusammenfassung Ordnung

Vier einfache Regeln, sieben gute Gründe:



1. Eins-zu-n Beziehungen: *eins unten, n darüber*
2. Eins-zu-(0..)eins Beziehungen: *auf gleicher Höhe*
3. n-zu-m Beziehungen: *auf gleicher Höhe, immer aufgelöst mit Klasse darüber*
4. Ableitungen: *Basisklasse unten, abgeleitete darüber*

Warum das Datenmodell so wichtig ist

In einer Schweizer Grossbank gilt schon lange die Faustregel:

2 : 5 : 20

2 Jahre hält das GUI

5 Jahre hält die Geschäftslogik

20 Jahre halten die Daten

Änderungen in der zentralen Datenhaltung (in der Datenstruktur) führen immer zu sehr weit gestreuten Änderungen im Code, von Geschäftslogik bis GUI. Alle stöhnen. Zu Recht.

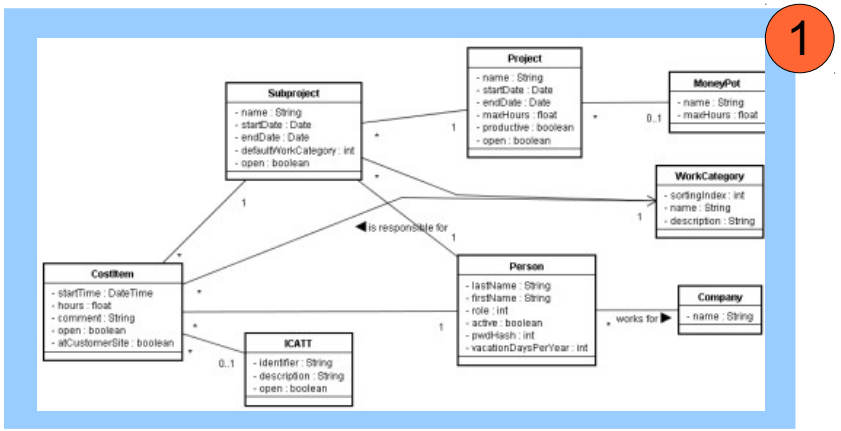
Domain- und Datenmodellierung

1. Schritt: Domainmodell
   - mit Operationen (Methoden)
   - mit evtl. überflüssigen Entitäten, evtl. unnötigen 1:1 Beziehungen...

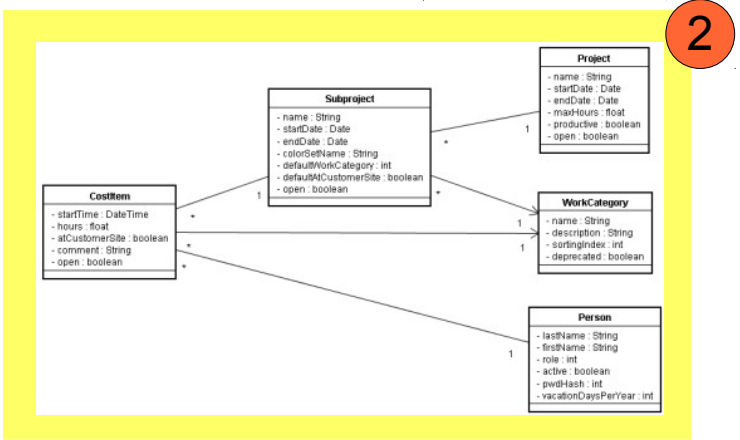
2. Schritt: Datenmodell (≈ ER Diagramm)
   - reine Daten-Objekte, ohne Methoden
   - immer 3NF, vorzugsweise DAG (s. später)
   - überflüssige 1:1 Beziehungen weg; weniger Vererbung

3. Schritt: Design-Klassenmodell
   - mit Data Model als eigene, separate Schicht

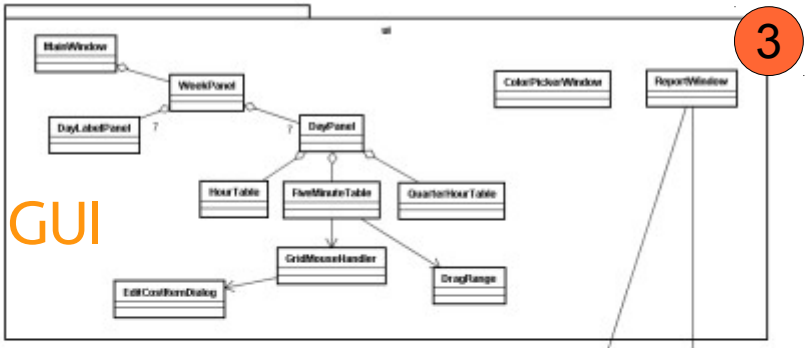
Architektur-Entwurfsmuster

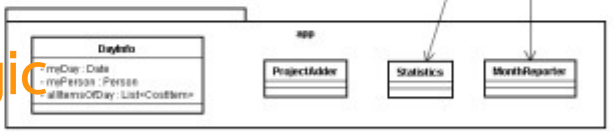


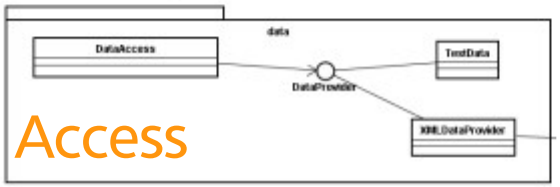
Domain-Modell

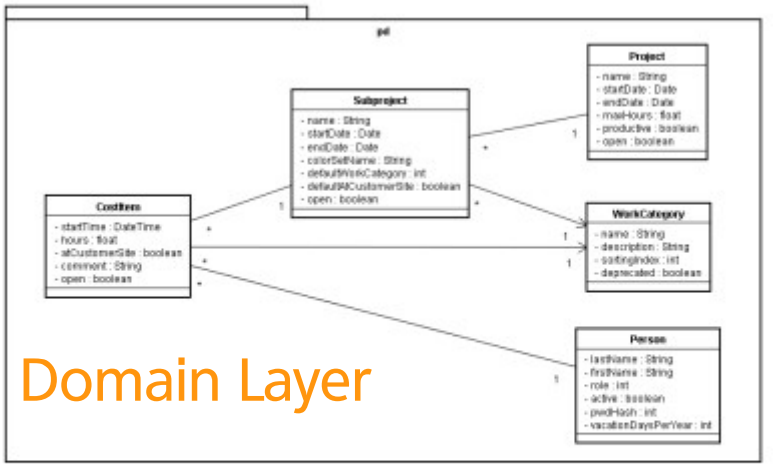


Datenmodell -› 1:1 als eigene Schicht





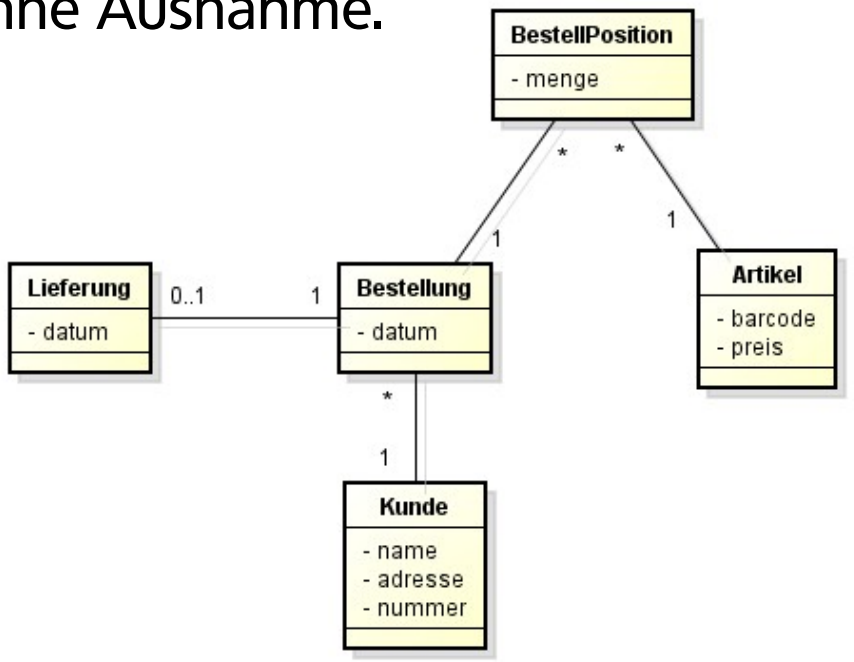




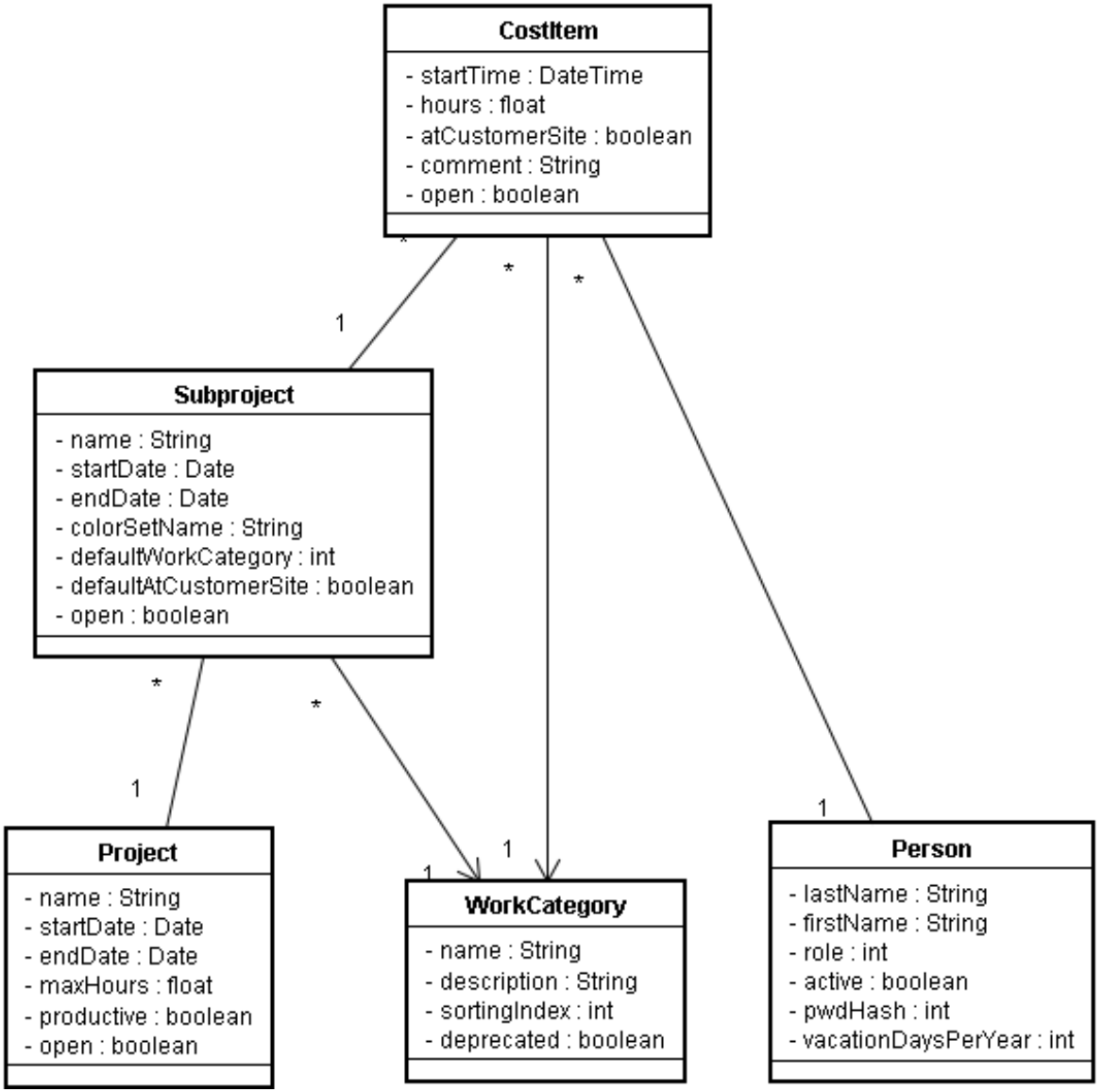
Empfehlungen zum Datenmodell

Beim Übergang vom Domain- zum Datenmodell:

- Klassen ohne Attribute sind meist nutzlos: weglassen.
- 1:1 Beziehungen hinterfragen: die sind meist nicht sinnvoll, daher eher zusammenführen.
  1: 0..1 Beziehungen dagegen können gut sinnvoll sein.
- n:m Beziehungen immer mit einer dazwischenliegenden (darüber positionierten) Klasse auflösen, ohne Ausnahme.



Daten-Modell in UML

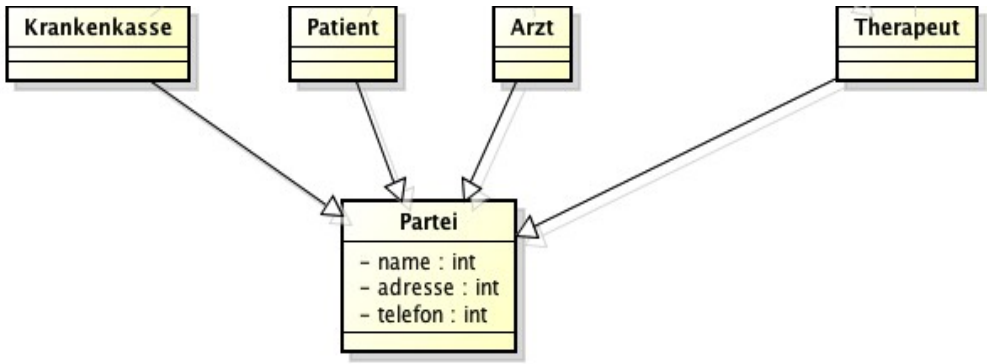


- Könnte auch in der Entity-Relationship Notation sein (ER Diagramm)
- Könnte auch das Domain Model sein (mit Kardinalitäten, aber ohne Operationen)
- Data Storage Classes, persistent classes
- Einfach "Datenbank" denken, auch wenn nie eine DB drunter kommt!

Empfehlungen zu Vererbung

Oft überbewertet, besser Delegation.

- Es kommt immer darauf an, wie man von eine Vererbung profitieren kann: z.B. Iteration zwecks Einladung über alle 'Personen', unabhängig davon, ob es 'Mitarbeiter, Student, Dozent, Gast' sind. Falls nichts derartiges gemacht wird, bringt der Vererbungs-Mechanismus nichts.
- Falls die abgeleiteten Klassen keine Attribute haben, muss man sich auch fragen, was die Vererbung soll.
- Wenn Vererbung, dann für die DB-Implementierung eher die Übermengen-Methode anwenden: nur EINE Klasse mit einem Typ-Attribut plus alle Attribute aller abgeleiteten Klassen.



Das Wichtigste nochmals in Kürze

- Schichten: oben/unten, asymmetrisch
- Begriffe: Schicht/Layer, Partition, Tier
- Regeln für Zugriffe („nie von unten nach oben, nicht quer…“)
- Separation of Concerns
- Datenmodell ordnen! (1 unten, n oben)
- 2 : 5 : 20
- Architektur-Entwurf: GUI – App – Data Access – Domain - Services

Wozu Domainmodellierung?

- Analyse der Requirements: präzise, bringt Unklarheiten hervor.
- Basis für Datenmodell und evtl. daraus erzeugte DB-Struktur.
- Vergleich mit anderen Projekten, u.a. zur Umfangabschätzung.
- Datenmodell 1:1 in Architekturschicht schieben: Lösung fertig!
- Dokumentation „Das haben wir umgesetzt“.