https://www.youtube.com/watch?v=283EGmUxRN0

Warum empfehle ich Ihnen nicht "ein Buch" für diese Veranstaltung?

Teil 1

Gute Bücher, die man nicht unbedingt besitzen muss

- Es gibt leider nicht **das** gute Software-Engineering-Buch ☹
- Folgende Buch-Typen findet man: (abgewandelt nach nach Siedersleben, J. (2004). *Moderne Software-Architektur: Umsichtig planen, robust bauen mit Quasar* (1st ed.). dpunkt, S. 12)

1. **Softwaretechnik-Klassiker**: große, umfangreiche „Standard-Lehrwerke“
2. **Theorie-Werke** aus der universitären Forschung, ohne viel Rückhalt in der Praxis
3. Bücher zu **Patterns** / Entwurfsmustern / Architekturstilen
4. **Technik-Bücher** (z.B. zu JEE, AngularJS, …)
5. **gute Ratschläge** (Ratgeber zu bestimmten Detailthemen)

1) Softwaretechnik-Klassiker

- Große, umfassende “Standardwerke”, die alle Aspekte des SW-Lebenszyklus abdecken. I.d.R. sehr umfangreich (800+ Seiten).
- Oft recht prozess- und standardlastig. Hilft eher nicht bei der Frage “Was tue ich als erstes, zweites, drittes, wenn ich ein SW-System neu erstellen will”.
- Gut geeignet als Nachschlagewerk – jedenfalls, wenn man in großen Organisationen unterwegs ist und bestimmte Standards beim Vorgehen und der Dokumentation einhalten muss. Dann kann man dort gut einmal nachschlagen, um den Aspekt besser verstehen und einordnen zu können.
- **Folgende Beispiele gibt es hier zu nennen (kein Anspruch auf Vollständigkeit!)**

- Klassische Standardwerke zu Software Engineering im Ganzen
  - Balzert, Helmut. *Lehrbuch der Softwaretechnik: Entwurf, Implementierung, Installation und Betrieb*. 3. Aufl. 2012. Heidelberg: Spektrum Akademischer Verlag, 2011. (Es gibt mehrere Bände aus dieser Reihe)
  - Sommerville, I. (2016). *Software Engineering* (10. Auflage). Pearson Studium.
  - Booch, Grady, Robert A. Maksimchuk, Michael W. Engle, Bobbi J. Young, Jim Conallen, and Kelli A. Houston. *Object-Oriented Analysis and Design with Applications*. 3rd ed. The Addison-Wesley Object Technology Series. Upper Saddle River, NJ: Addison-Wesley, 2007.
  - Jackson, Michael. *Problem Frames & Methods: Analysing and Structuring Software Development Problems*. Harlow: Addison-Wesley Longman, Amsterdam, 2000.

1) Softwaretechnik-Klassiker (Forts.)

- Empfehlenswerte Standardwerke speziell zu Architekturthemen (nur Beispiele)
  - Capgemini. (2015). Architecture Guidelines for Application Design v2.0.
  - Dustdar, Schahram, Harald Gall, and Manfred Hauswirth. *Software-Architekturen für Verteilte Systeme: Prinzipien, Bausteine und Standardarchitekturen für moderne Software*. 1st ed. Springer Berlin Heidelberg, 2003.
  - Gregor Engels, Andreas Hess, Bernhard Humm, Oliver Juwig, Marc Lohmann, Jan-Peter Richter, Markus Voß, and Johannes Willkomm. *Quasar Enterprise: Anwendungslandschaften serviceorientiert gestalten*. 1., Aufl. dpunkt.verlag, 2015.
  - Lilienthal, Carola. *Langlebige Software-Architekturen: Technische Schulden analysieren, begrenzen und abbauen*. 1st ed. dpunkt.verlag GmbH, 2015.
  - Newman, S. (2015). *Building Microservices* (1st ed.). O'Reilly and Associates.
  - Siedersleben, Johannes. *Moderne Software-Architektur: Umsichtig planen, robust bauen mit Quasar*. 1st ed. Heidelberg: dpunkt, 2004.
  - Wolff, Eberhard. *Microservices: Grundlagen flexibler Softwarearchitekturen*. 1., Auflage. dpunkt.verlag, 2015.

- Fokus auf Modellierung mit UML
  - Als Referenz sollten Sie immer in den frei verfügbaren Standard schauen:
    - OMG. "OMG Unified Modeling Language, Version 2.5.1," 2015. https://www.omg.org/spec/UML/2.5.1/PDF
  - Darüber hinaus gibt es eine Vielzahl von Büchern dazu (keine Empfehlung von mir an dieser Stelle)

2) Ergebnisse der akademischen Forschung

?

- In der Softwareentwicklung scheinen alle nennenswerten Innovationen der letzten zwei Jahrzehnte **NICHT** aus der universitären Forschung gekommen zu sein.
  - Das ist schon ungewöhnlich – und bei Medizin, Chemie, Maschinenbau, … etc. anders. Auch in anderen Bereichen der IT (z.B. KI, Data Science) gilt das sicher nicht.
  - In der Softwareentwicklung ist eher die Community von anerkannten Expert*innen der Innovationstreiber.
  - Daher keine Bücher hier aufgeführt!

- **Einzige Ausnahme, die mir einfällt:** Dissertation von Roy Fielding, die die Grundlage von REST darstellt
  - Fielding, R. T. (2000). *Architectural styles and the design of network-based software architectures* [University of California, Irvine]. http://jpkc.fudan.edu.cn/picture/article/216/35/4b/22598d594e3d93239700ce79bce1/7ed3ec2a-03c2-49cb-8bf8-5a90ea42f523.pdf

3) Bücher zu Patterns / Entwurfsmustern / Architekturstilen

- Patterns haben sich als „Stilelement“ im Design von Software durchgesetzt. Es gibt eine Menge guter Bücher dazu. Man findet allerdings auch viele sinnvolle Informationen in Onlinequellen.
- Der Klassiker ist das „Gang of Four“ Buch:
  - Gamma, Erich, Richard Helm, Ralph E. Johnson, and John Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software.* 1st ed., Reprint. Reading, Mass: Prentice Hall, 1994.

- Weitere empfehlenswerte Werke dazu:
  - Eilebrecht, Karl, and Gernot Starke. *Patterns kompakt: Entwurfsmuster für effektive Software-Entwicklung*. 4th ed. Berlin: Springer Vieweg, 2013.
  - Goll, Joachim. *Architektur- und Entwurfsmuster der Softwaretechnik*. Wiesbaden: Springer Fachmedien Wiesbaden, 2014. http://link.springer.com/10.1007/978-3-658-05532-5.
  - Hohpe, Gregor, and Bobby Woolf. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. 01 ed. Boston: Addison Wesley, 2003.
    - Auch als frei verfüglicher, sehr empfehlenswerter Blog, der die wesentlichen Informationen beinhaltet: http://www.enterpriseintegrationpatterns.com/index.html.

4) Technik-Bücher (z.B. zu JEE, AngularJS, …)

- Hier gebe ich Ihnen keine Empfehlungen – schauen Sie selbst, was Sie brauchen. Der Markt ist viel riesig, und in dieser Veranstaltung geht es nicht um eine bestimmte Technologie (auch wenn wir Technologien vorgeben).
- Dieser Typ von Buch veraltet i.a. schnell. Man wird so ein Buch, wenn man es sich kauft oder aus der Bibliothek ausleiht, vermutlich einmal mehr oder weniger komplett lesen, später aber höchstens noch als Referenz nutzen.
  - Da gibt es dann gute Onlinequellen (z.B. Stackoverflow) als Konkurrenz.

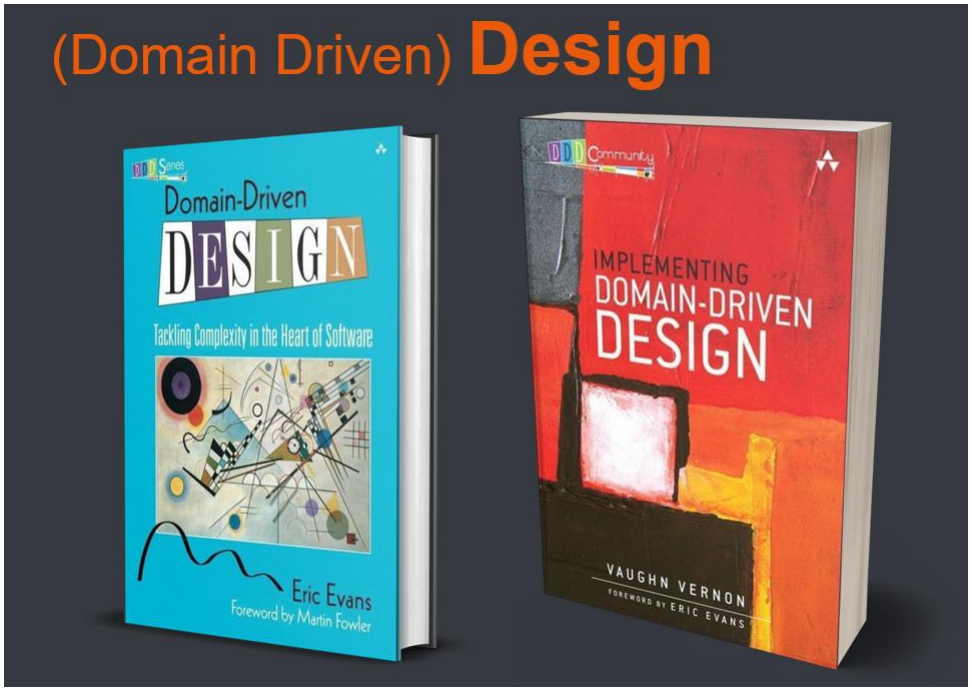
5) gute Ratschläge (Ratgeber zu bestimmten Detailthemen)

- “Ratgeber” sind Bücher zu bestimmten Aspekten des Entwicklungsprozesses.
- Meist liest man sie von vorn bis hinten (vielleicht unter Auslassung bestimmter Kapitel), weil der im Ratgeber behandelte Aspekt gerade von besonderer Bedeutung ist.
- Solche Bücher kann man später immer wieder als Nachschlagewerke nutzen.

- Beispiele für gute Ratgeber
  - Hruschka, P., & Starke, G. (2014). *Knigge für Softwarearchitekten—Reloaded*. entwickler.press.
  - Toth, Stefan. *Vorgehensmuster für Softwarearchitektur: Kombinierbare Praktiken in Zeiten von Agile und Lean*. 2., aktualisierte und erweiterte Auflage. München: Carl Hanser Verlag GmbH & Co. KG, 2015.
  - Zörner, Stefan. *Softwarearchitekturen dokumentieren und kommunizieren: Entwürfe, Entscheidungen und Lösungen nachvollziehbar und wirkungsvoll festhalten*. 2., Aufl. München: Carl Hanser Verlag, 2015.
- Für ST2 relevant: REST-Ratgeber
  - Massé, M. (2011). *REST API Design Rulebook* (1. Aufl.). Beijing: O’Reilly and Associates.
  - Tilkov, Stefan, Martin Eigenbrodt, Silvia Schreier, and Oliver Wolf. REST und HTTP: Entwicklung und Integration nach dem Architekturstil des Web, p. 11ff. 3. Aufl. Heidelberg: dpunkt.verlag GmbH, 2015.

- Während die vorgenannten Arten von Büchern meist doch nur punktuell benötigt werden, sind die nachfolgenden vier Bücher (nach Meinung die Autors!) von einer bleibenden Qualität – man kann sich dort immer wieder Anregungen holen.
- Für die Veranstaltung ST2 werden diese vier Bücher eine größere Rolle spielen.

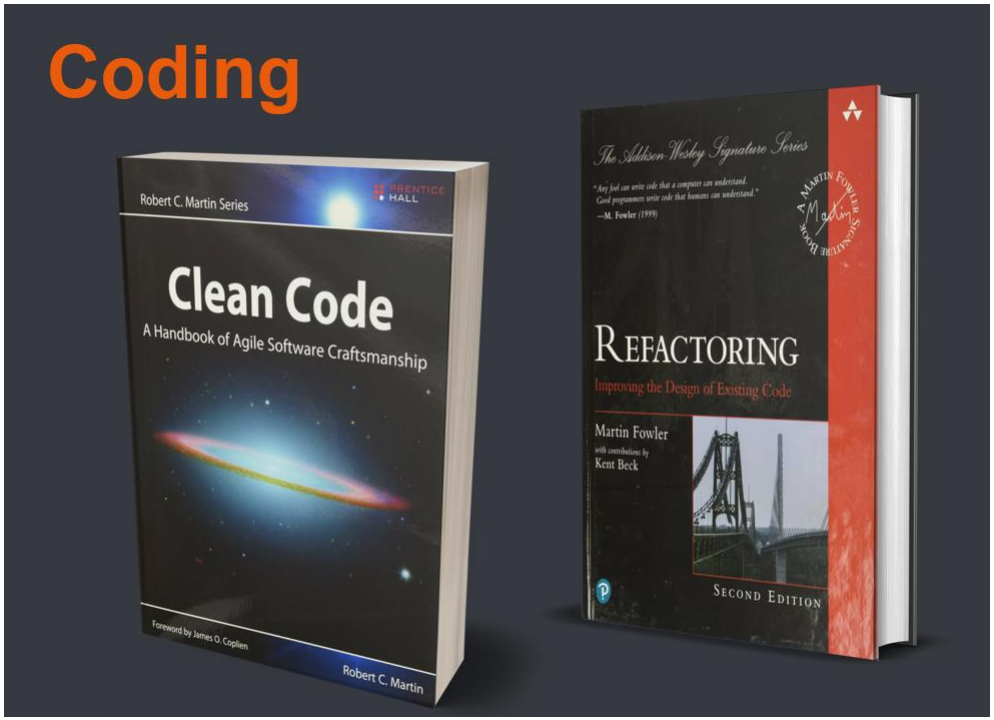
Domain-Driven Design



- DDD wurde in dieser Veranstaltung bereits ausführlich eingeführt.
- Mit diesen beiden Büchern hat man eine umfassende Grundlage zu diesem Designansatz.
- Evans (“blaues Buch”) ist gut für die Grundlagen, aber etwas sperrig zu lesen.
- Vernon (“rotes Buch”) liest sich flüssiger und ist gut als aufbauende Fortsetzung zu Evans zu lesen.

- Evans, E. (2003). *Domain-Driven Design: Tackling Complexity in the Heart of Software* (1 edition). Addison-Wesley Professional.
- Vernon, V. (2013). *Implementing Domain-Driven Design* (1 ed.). Addison Wesley.

Coding



- Bob Martin (a.k.a. “Uncle Bob”) hat diesen modernen Klassiker als Beschreibung einer “handwerklich sauberen” Haltung beim Coden geschrieben.
- Martin Fowlers Buch beschreibt (in gewisser Weise aufbauend darauf), wie man technische Schuld (schlechten Code, schlechte Architektur) Stück für Stück besser macht.
- Die Inhalte beider Bücher werden in der ersten Phase von ST2 (Meilenstein M1) angewendet.

- Martin, R. C. (2008). *Clean Code: A Handbook of Agile Software Craftsmanship* (1st ed.). Prentice Hall.
- Fowler, M. (2018). *Refactoring: Improving the Design of Existing Code* (2nd ed.). Addison-Wesley Professional.

Fachliches Datenmodell

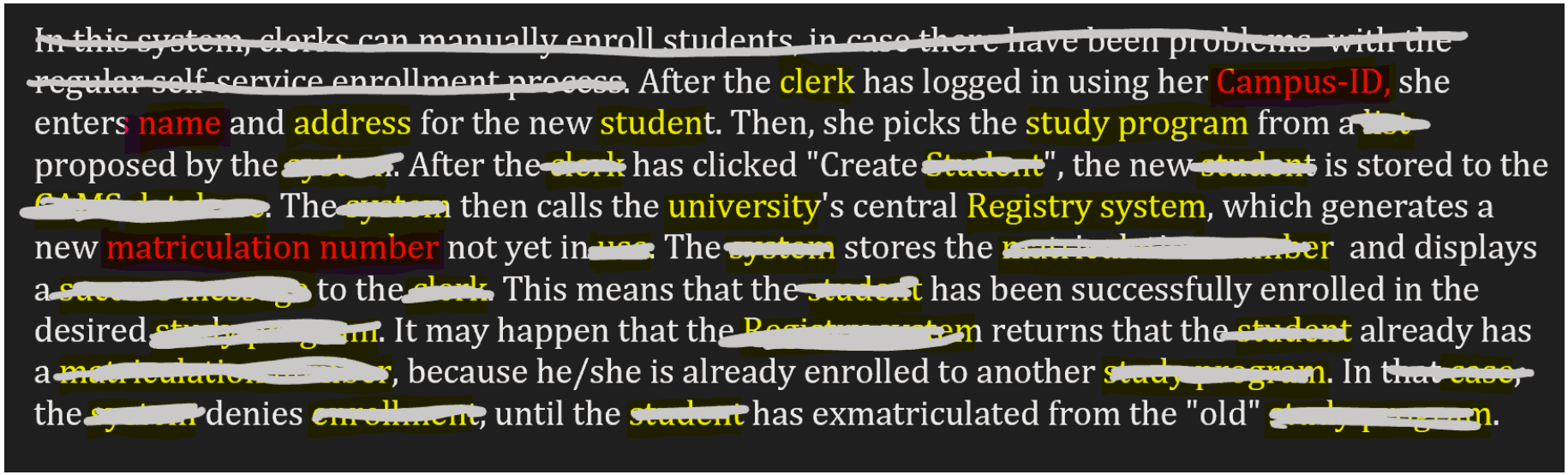
als einfaches UML-Klassendiagramm

Technology
Arts Sciences
TH Köln

https://www.youtube.com/watch?v=-xlb9olccHI

Was ist ein fachliches Datenmodell?

- Das fachliche Glossar erklärt die Geschäftsobjekte
- **Beziehungen** zwischen den Geschäftsobjekten sind noch nicht beschrieben
- => Wir haben einfach nur eine Liste von Begriffen

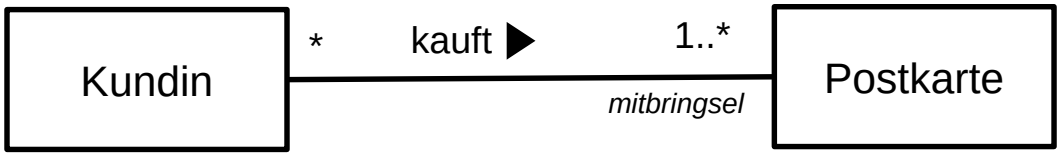


Reale Welt:

Kundin kauft Postkarte als Mitbringsel

Fachliches Datenmodell:

bringt die Begriffe in Beziehung

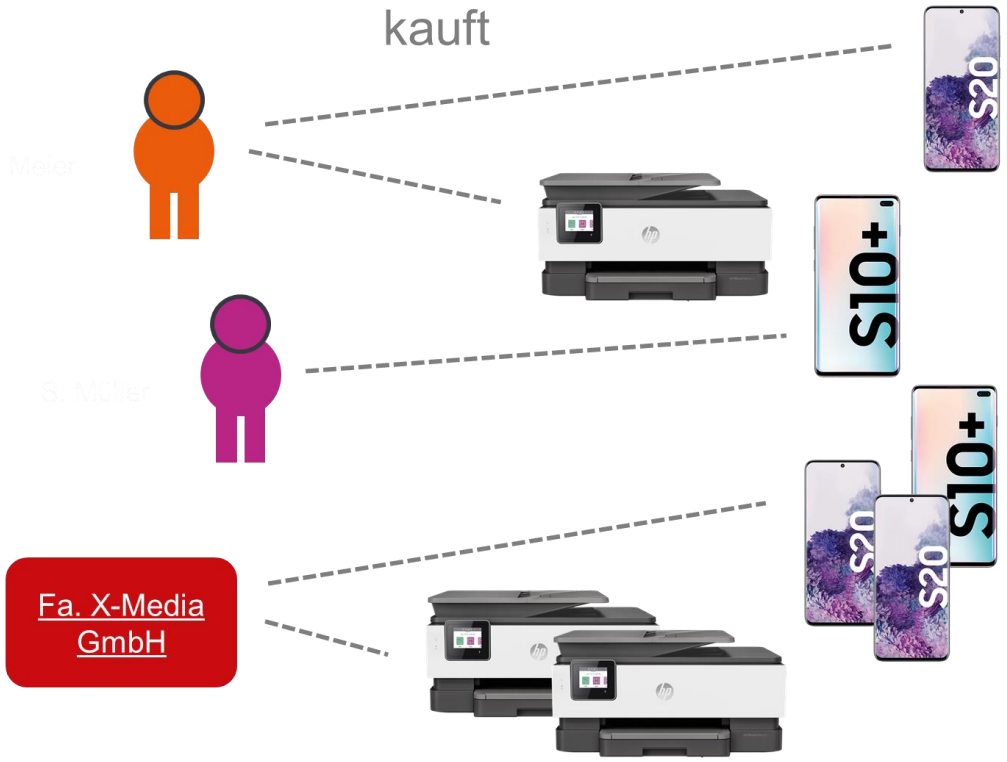


Fachliches DM als einfaches UML-Klassendiagramm

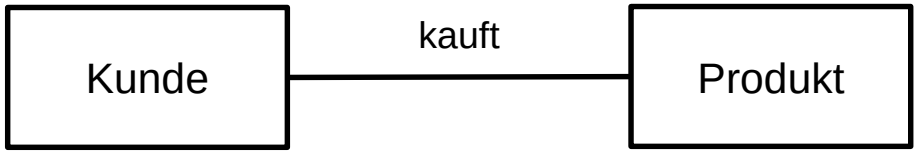
- Klassendiagramme: am häufigsten genutztes UML-Diagramm
- Beschreibt …
  - Klassen von Objekten
  - Deren Beziehungen zu einander
- Jetzt nur die für das fachliche DM nötigen Eigenschaften
  - Einfache Klassen
  - Attribute in besonderen Fällen
  - Ungerichtete Beziehungen
    - ggfs. mit Namen
  - Multiplizitäten
  - Vererbung
  - Komposition
- Weitere Eigenschaften später

Unterschied Objekt - Klasse

- **Objekte** eines Typs

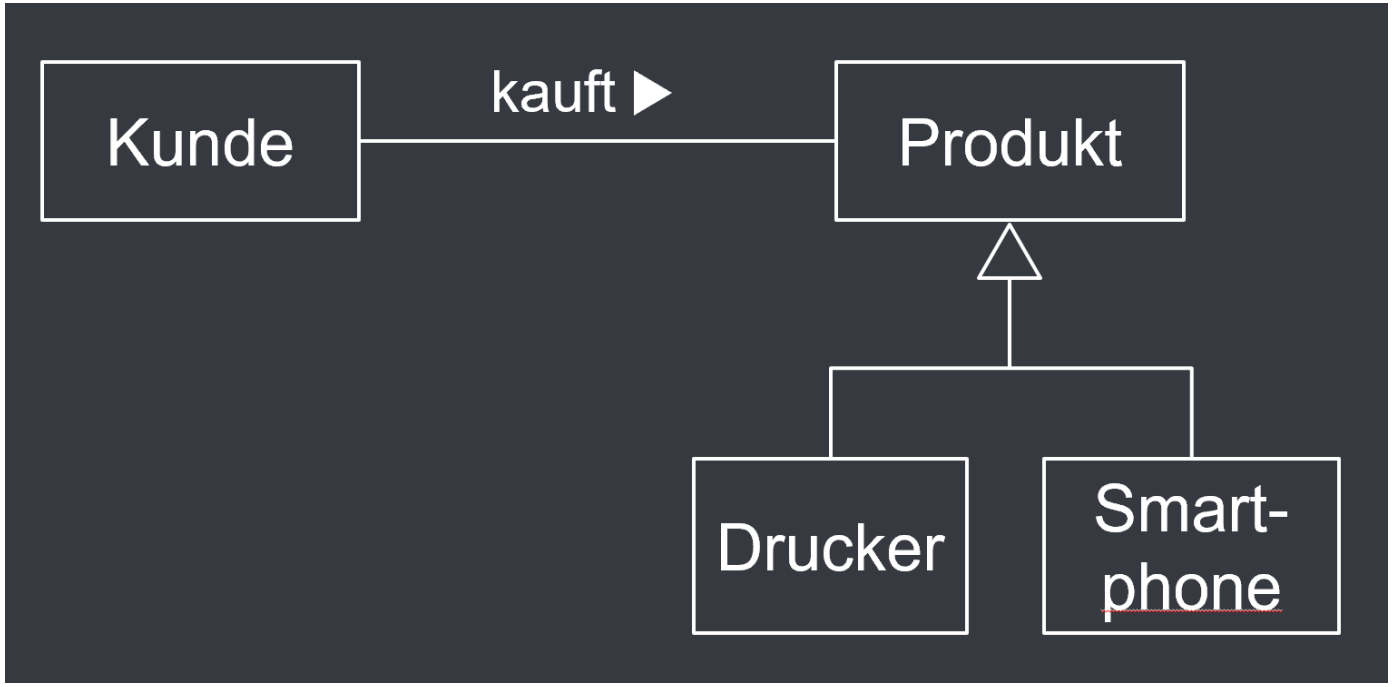


- **Klassen**
- Zusammenfassung von Objekten mit
  - gleichen Merkmalen
  - gleichen Einschränkungen
  - gleicher Semantik



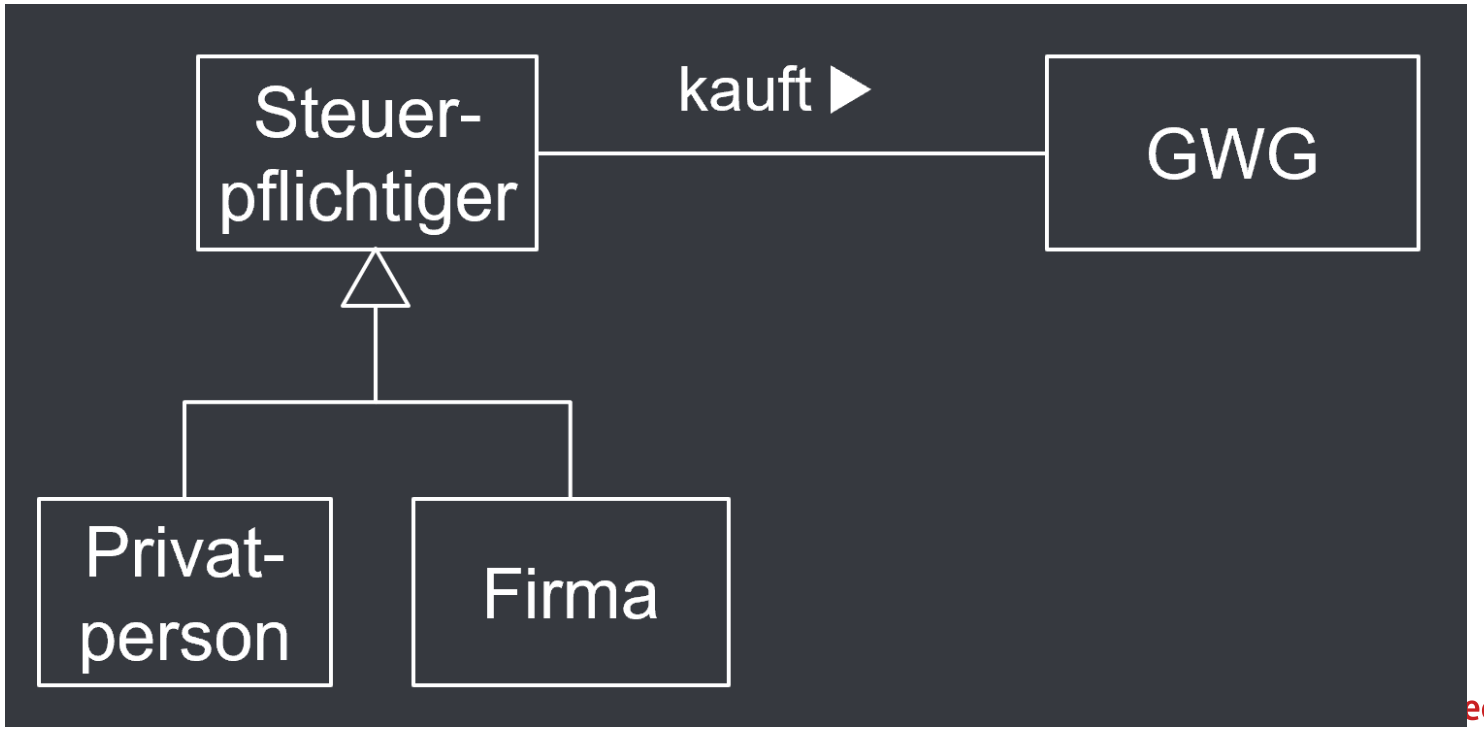
Welche Klassen gewählt werden, hängt vom Kontext ab (1)

- Wenn der Kontext z.B. ein **Webshop** ist, dann ist nur wichtig, dass ein Kunde ein Produkt kauft.
- Wichtig ist dann noch die Unterscheidung nach den verschiedenen Produktarten: Für den Webshop sind damit wichtige Unterschiede verbunden (z.B. andere Werbung, Art der Verpackung, ggfs. verschiedene Mehrwertsteuer, …)



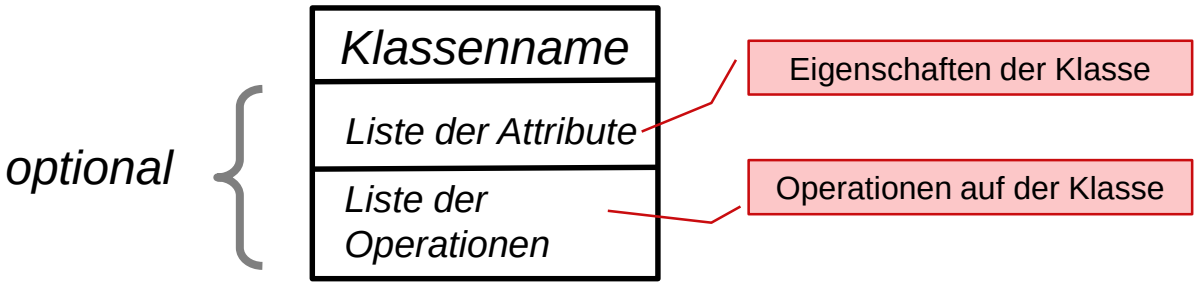
Welche Klassen gewählt werden, hängt vom Kontext ab (2)

- Ist der Kontext aber z.B. ein **Steuerberater**, dann sieht der Kontext vielleicht ganz anders aus.
- Hier ist der gekaufte Gegenstand nicht so wichtig: Alles unter 800€ ist ein **geringwertiges Wirtschaftsgut (GWG).**
- Wichtig ist aber der zu besteuernde Käufer: Eine Firma wird den Kauf steuerlich ganz anders absetzen als eine Privatperson.
- **Dieselbe Realwelt-Situation führt zu einem völlig anderen fachlichen Datenmodell!**

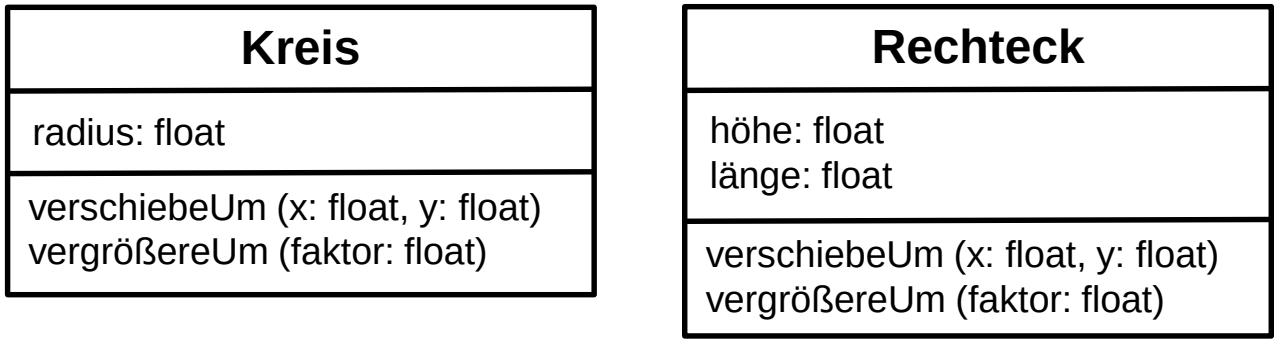


Attribute und Operationen einer Klasse

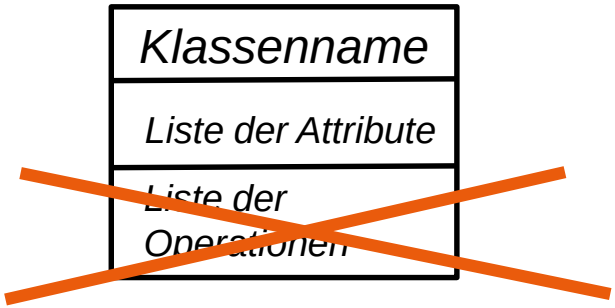
- Das UML-Klassendiagramm wurde ursprünglich entwickelt, um objektorientierte Klassen zu modellieren



- Klassisches Beispiel: Implementierung eines grafischen Editors



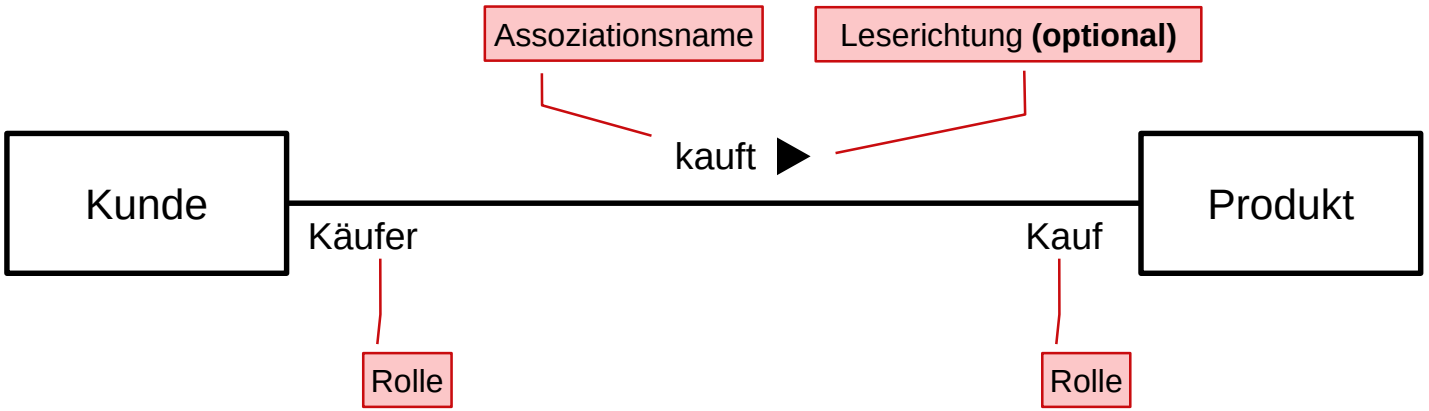
Fachliches Datenmodell: keine Operationen



- Operationen werden im fachlichen Datenmodell nicht benutzt.
- Grund: Die Verhaltensperspektive unserer Domäne bilden wir mit anderen, geeigneteren Mitteln ab (Use Cases, Statusdiagramme, Aktivitätsdiagramme, etc.)

Assoziationen

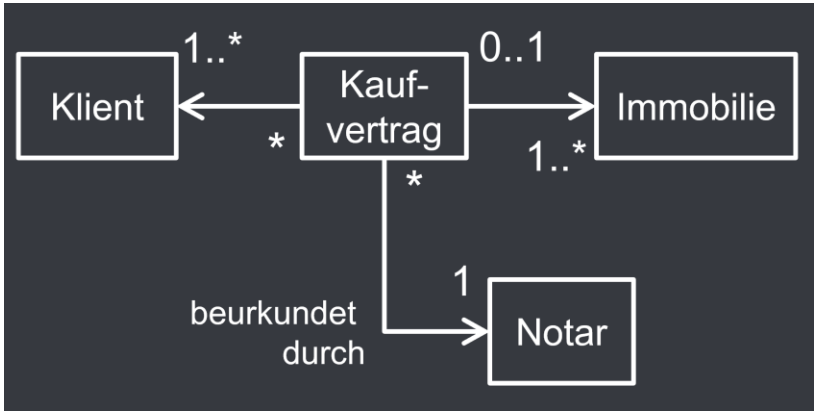
- Beschreibt die gemeinsame Struktur einer Menge von statischen Beziehungen zwischen Objekten
- Aussagekraft kann erhöht werden durch optionale Angabe von
  - Assoziations-Name, ggfs. mit Leserichtung
  - Rollen
  - Multiplizitäten
  - *(weitere später)*



Fachliches Datenmodell: immer ungerichtete Assoziation

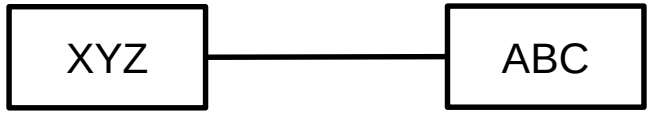
- Im **fachlichen Datenmodell** setzt man immer eine **ungerichtete** Assoziation ein. Diese hat keine Pfeilspitzen und legt keine Richtung der Beziehung nahe.

- Später, im **logischen Datenmodell** (kommt in Teil 2 dieser Veranstaltung), ändert sich das. Im logischen Datenmodell bereiten wir die Umsetzung in Code + Datenhaltungs-Struktur vor.
- Da müssen wir dann wissen, von wo nach wo Referenzen existieren. Oder mit anderen Worten: Welches Objekt „kennt“ das andere Objekt, mit dem es in Beziehung steht?
- Gerichtete Assoziationen werden über **Pfeile** ausgedrückt.
- Als Vorgriff auf Teil 2 sehen Sie rechts den entsprechenden Ausschnitt aus dem logischen DM. Man braucht für den Kauf auch einen Vertrag (Bewegungsdaten, => später mehr). Außerdem muss festgehalten werden, wer den Vertrag beurkundet.
- Da eine Immobilie und ein Klient in vielen verschiedenen Kontexten auftauchen können, ist „Vertrag“ das **speziellere Konzept** und verweist damit auf Klient und Immobilie.
- Näheres dazu dann beim logischen DM in Teil 2.

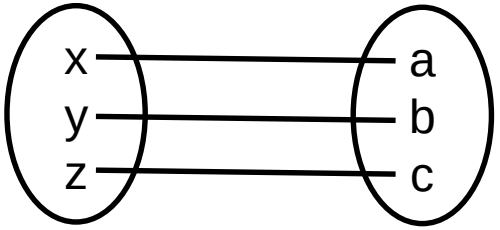


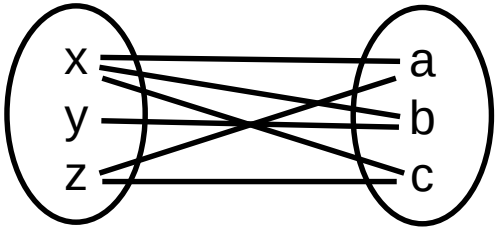
Multiplizitäten von Assoziationen

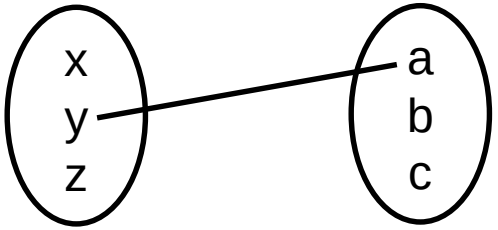
- Assoziation auf Klassen-ebene …



- … kann viele verschiedene Ausprägungen auf Objekte-ebene haben

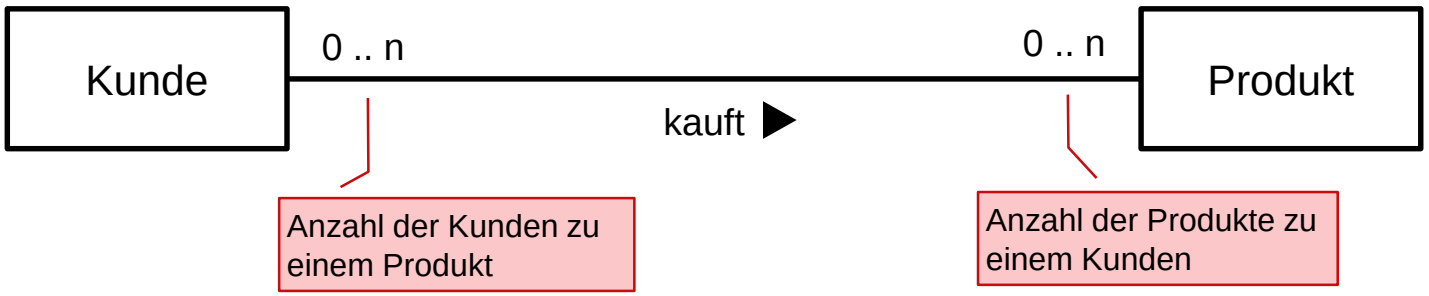






- Multiplizitäten einer Assoziation:
- die jeweils **möglichen** Anzahlen

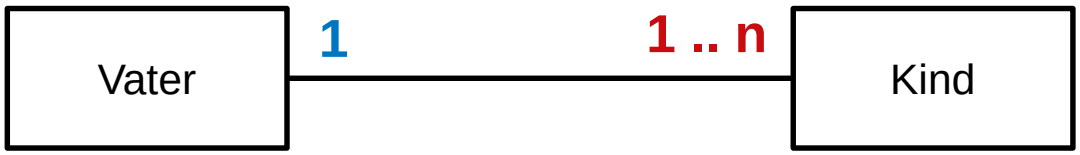
Multiplizitäts-Annotation im Diagramm

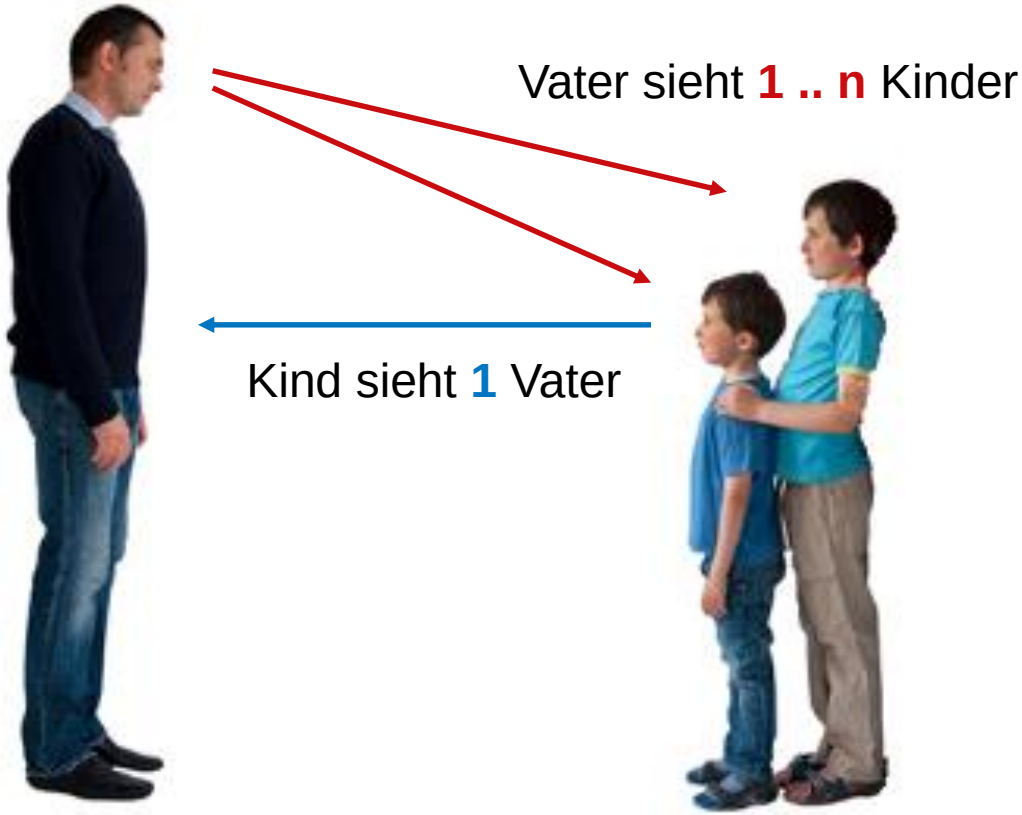


- `1` genau 1
- `0..1` keins oder eins
- `0..n`, `0..*`, `*` beliebig viele
- `1..n`, `1..*` mindestens 1

- `1,2,4` entweder 1, 2 oder 4
- `2..5` zwischen 2 und 5
- `0..2,4..*` alles, nur nicht 3

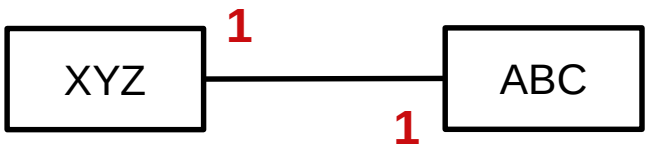
Eselsbrücke für Ort der Multiplizität: "Da wo man hinguckt"

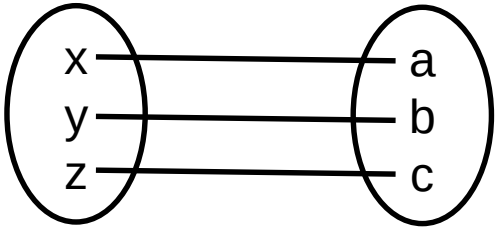


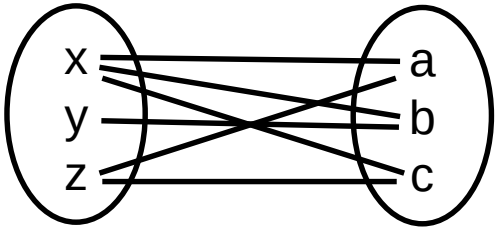


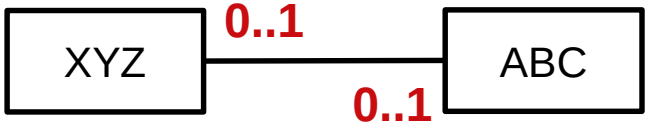
Multiplizitäten von Assoziationen

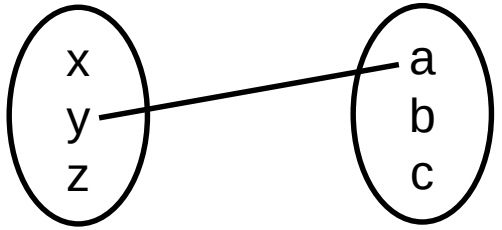
- Was beschreibt **am genauesten** die jeweils möglichen Anzahlen?
  - Achtung: `*` (`0..`n) ist so allgemein, dass es **immer** passt!



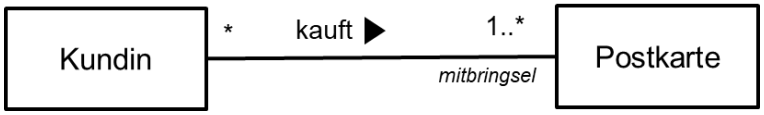




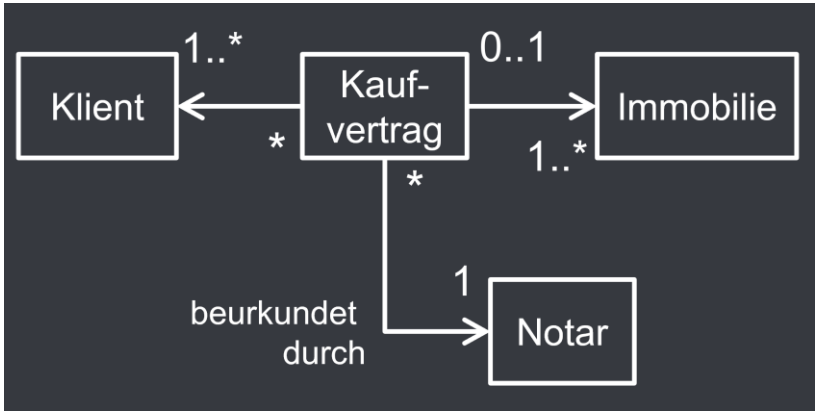




Zu den Multiplizitäten …



- **Aus einer studentischen Frage im Discord:**
- *Im Skript […] wird davon ausgegangen, dass Postkarten von mehreren Kunden gekauft werden können. Es handelt sich bei der Betrachtungsweise also nicht um speziell eine Postkarte. Die könnte ja nur von einer Person gekauft werden. […] bei der Beziehung zwischen Kaufvertrag und Notar wird aber davon ausgegangen das ein Kaufvertrag von genau einem Notar beurkundet wird. Das heißt doch hier wird von einem bestimmten Kaufvertrag ausgegangen. Sonst müsste man doch ein * einfügen, da Kaufverträge von Notaren beurkundet werden.*



- **Antwort:**
- Der Schlüssel liegt hier in der Interpretation der Klasse. Bei "Vertrag" ist das Objekt der eine spezifische Vertrag für Frau Müller und die Finca auf Mallorca. Bei "Postkarte" wären die Objekte eher "Postkarten-Motiv Katze", "Postkarten-Motiv Sonnenuntergang", etc. und weniger die einzelne physisch anfassbare Postkarte. Wenn Sie ein Warensystem programmieren, würden Sie vielleicht pro Postkarten(motiv) die Kunden nachhalten wollen (werden Katzenkarten eher von Frauen gekauft, etc. ...), aber Sie werden sicher nicht jede einzelne Postkarteninstanz als eigenes Objekt erfassen.
- (Gedankliche Übersetzung: Gibt es für jede Postkarte, die verkauft wird, eine DB-Tabellenzeile? Oder für jedes Postkartenmotiv, und dazu erfassen wir einen Bestand?)
- Durch diese Überlegung wird deutlich, dass die Postkarten-Situation vielleicht nicht so klar ist wie gedacht, und man vielleicht doch jeden Kaufvorgang erfassen muss. Dann hätten wir so etwas wie
  - | Kaufvorgang (Preis, Datum) | --- | Produkt |
  - von Produkt sind mehrere Klassen abgeleitet, wie z.B. | Postkarte(Motiv) |.
- Jede Diskussion mit der Fachseite (und sowas ähnliches will dies hier "simulieren") bringt wieder mehr Deutungsklarheit, und man muss seine Modellierung ggfs. überarbeiten. s

Spezielle Beziehungen: Generalisierung (Vererbung)

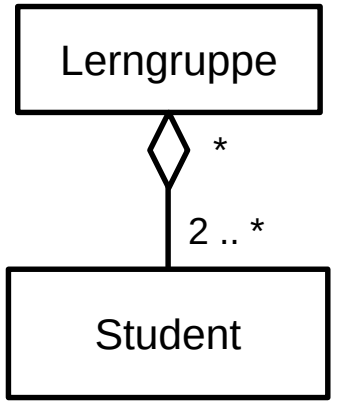
- *Taxonomische* Beziehung („*is-a*“) zwischen
  - spezialisierten Klasse (*Unter-, Subklasse*) und
  - einer allgemeineren Klasse (*Ober-, Super-, Basisklasse*)
- Subklasse übernimmt („*erbt*“) die Merkmale der Superklasse
  - Die Subklasse kann zusätzliche Merkmale enthalten

Quelle: nach [Jochum]

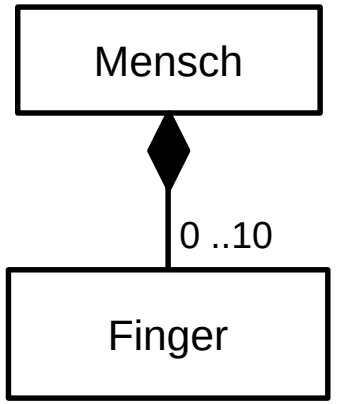
Spezielle Beziehungen: Aggregation, Komposition

- Zusammenfassung von Teilen zu einem Ganzen

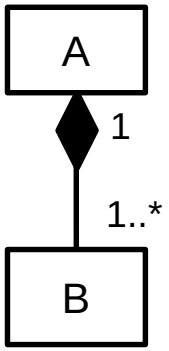
- **Aggregation**
  - Teile können losgelöst vom Ganzen existieren
  - Teile können zu mehreren Aggregaten gehören
  - Operationen nur z.T. von Ganzem auf Teile übertragen
    - (z.B. Kopieren ja, Löschen nein)



- **Komposition**
  - Teile können nur zu *einem* Ganzen gehören
  - Operationen auf Ganzem werden auf Teile übertragen
- Daumenregel: "Wenn ich das Ganze lösche, dann ergeben die Teile keinen Sinn mehr"



Häufiges Missverständnis zu Komposition



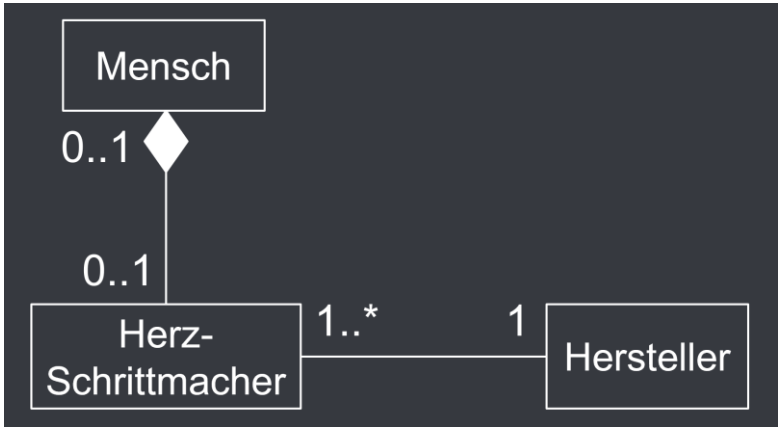
- Häufig höre ich als Begründung für die Wahl von **Komposition** in mündlichen Prüfungen:
- „Ein Objekt b vom Typ B kann ohne ein a vom Typ A nicht existieren“.

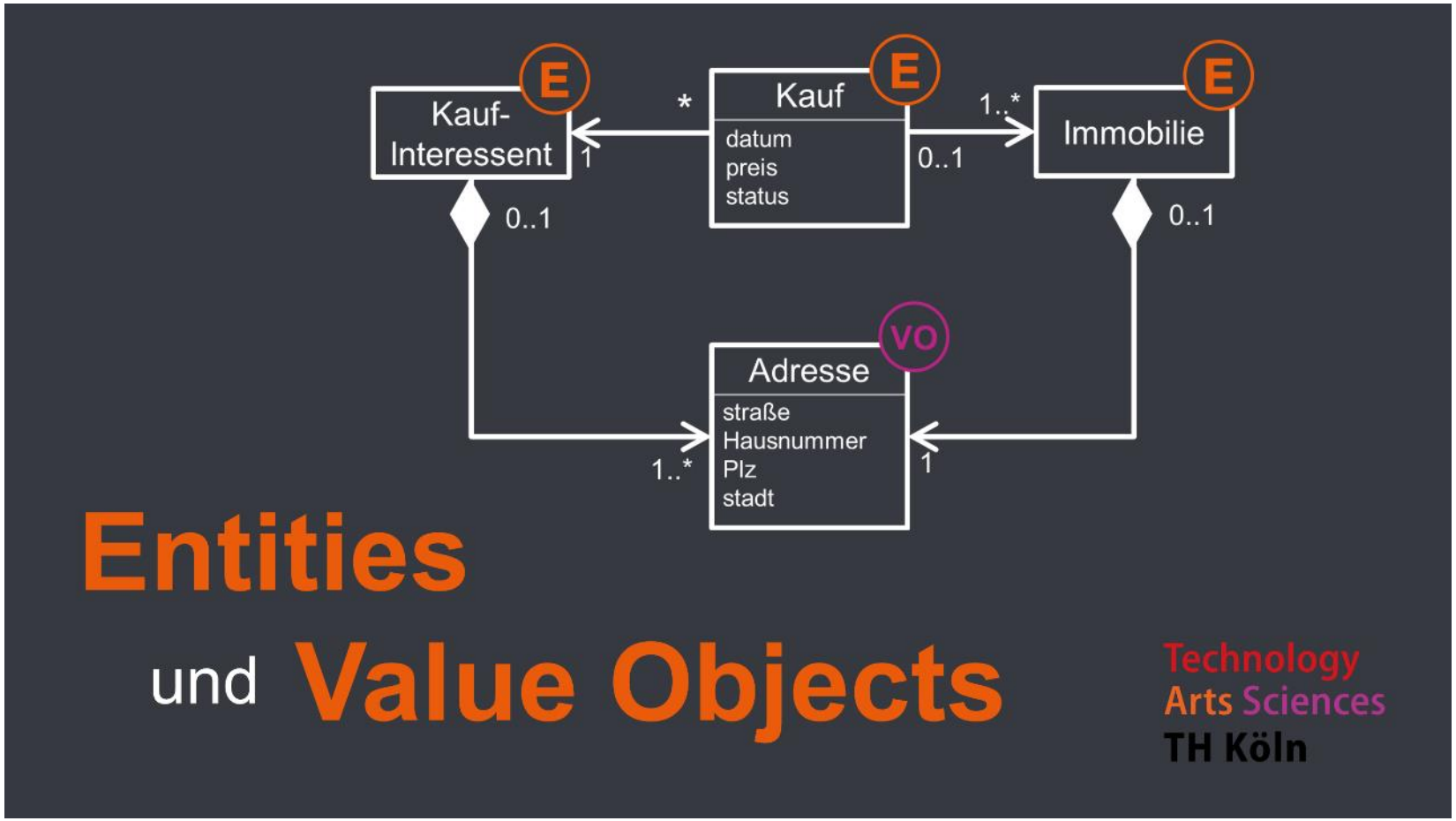
- **Die Begründung stimmt aber so nicht.**

- Im oberen Fall ist das tatsächlich so, aber das liegt an den **Multiplizitäten**, nicht an der Komposition: zu B gehört zwingend immer genau 1 A.

- Nehmen wir aber stattdessen folgendes Beispiel:
- Ein (älterer) Mensch kann einen Herzschrittmacher haben. Herzschrittmacher haben einen Hersteller und sind nach der Herstellung erst einmal natürlich noch nicht in einen Patienten eingesetzt.
- Obige Annahme greift also nicht: Ein Herzschrittmacher existiert auch außerhalb eines Menschen.
- Stattdessen gelten die beiden Begründungen für eine Kompositionsbeziehung, die auf der letzten Folie genannt wurden:

1. Verstirbt der ältere Mensch, wird sein Schrittmacher mit ihm bestattet. (Etwas makabres Beispiel, sorry, aber wir haben technisch gesehen ein kaskadierendes Löschen.)
2. Ein Schrittmacher kann immer nur zu höchstens einem Menschen gehören, niemals zu mehreren.

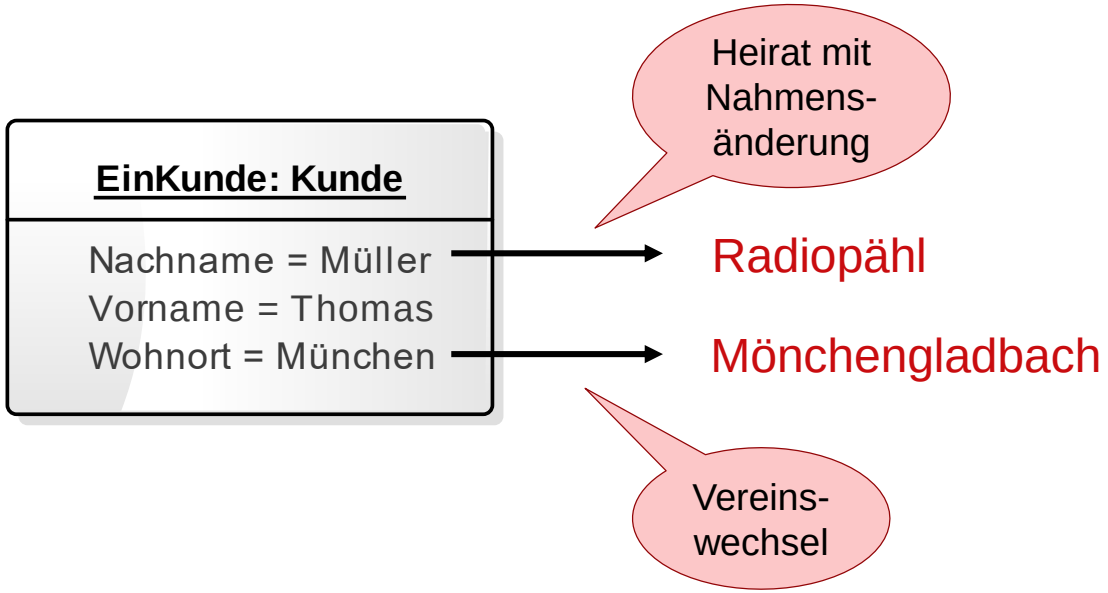




https://www.youtube.com/watch?v=pYg7I3UJ3Ls

Entitäten (Entities)

- **Entities** (Entitäten): Objekte mit unveränderlicher Identität
  - auch wenn Attributwerte sich ändern, bleibt Objekt "es selbst"



Bilder:
- https://www.unitedcharity.de/PROMINENTE/Thomas-Mueller
- Sascha Kohlmann, CC BY-SA 2.0, https://www.flickr.com/photos/skohlmann/8844560880

Entities haben Identität, auch bei gleichen Attributwerten

EinKunde: Kunde

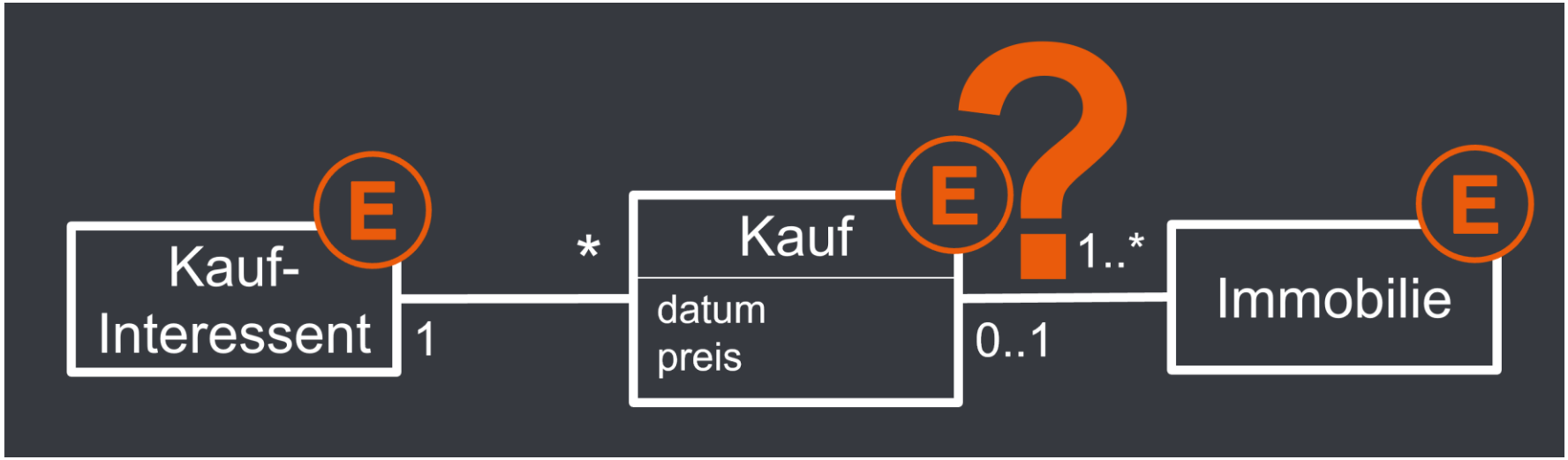
Nachname = Müller
Vorname = Thomas
Wohnort = München

NochEinKunde: Kunde

Nachname = Müller
Vorname = Thomas
Wohnort = München

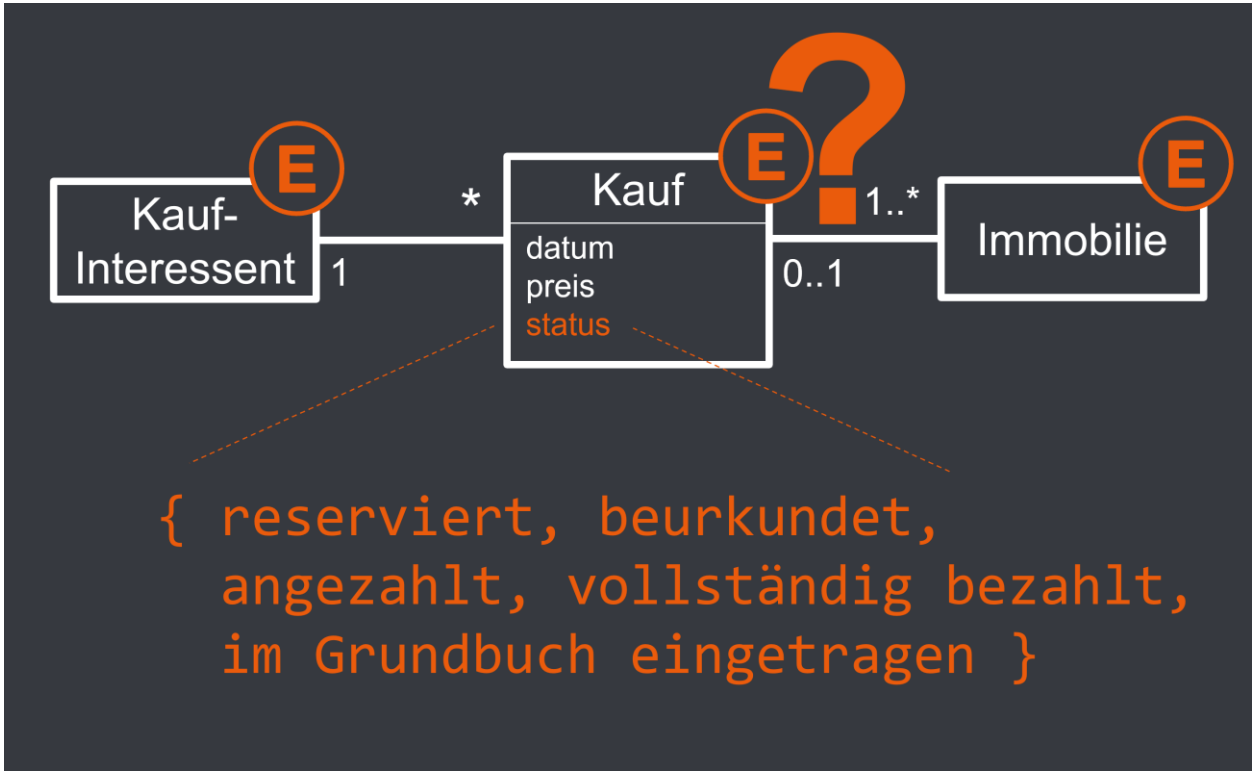
Bilder:
- https://www.unitedcharity.de/PROMINENTE/Thomas-Mueller
- Sascha Kohlmann, CC BY-SA 2.0, https://www.flickr.com/photos/skohlmann/8844560880

Ist alles ein Entity?



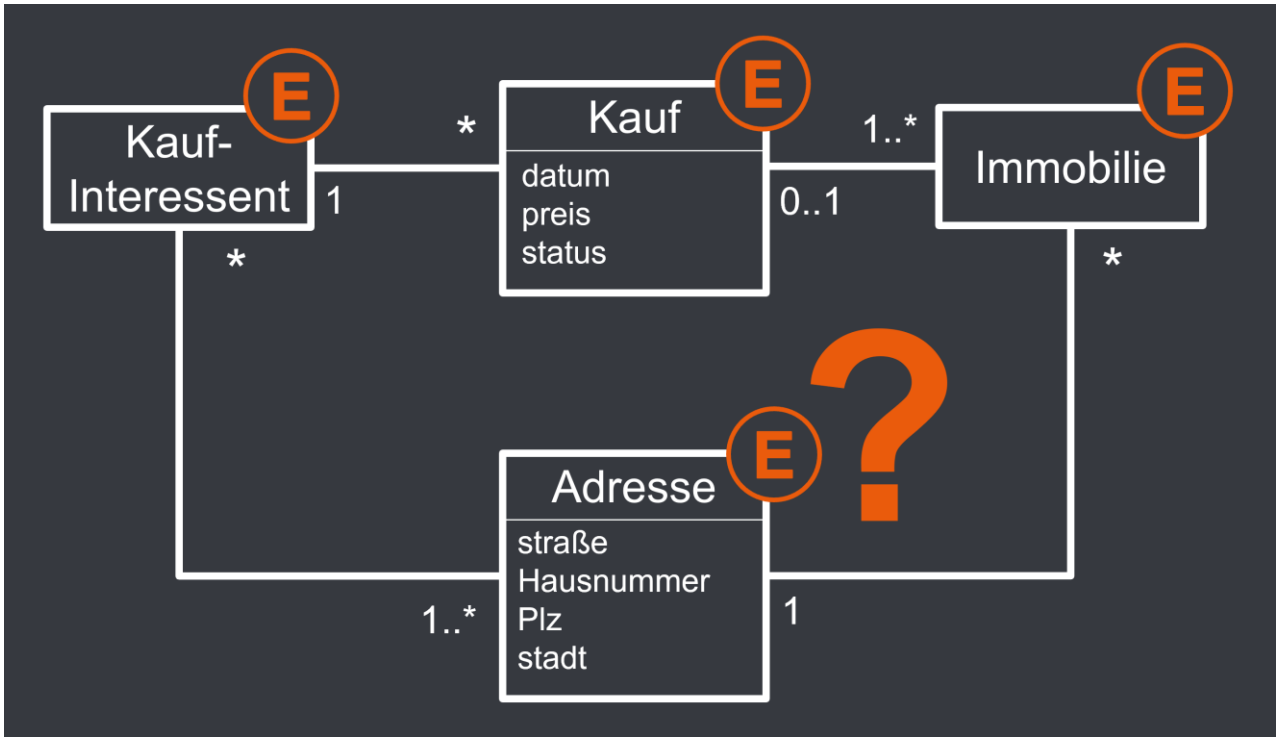
- **KaufInteressent** ist eine Person. Aus dem Beispiel von eben kann man ableiten, dass das ein Entity ist.
- Gleiches gilt auch für eine **Immobilie**. Ich kann z.B. eine Garage an die Immobilie anbauen. Dadurch verändern sich Attributwerte wie “Größe”. Es bleibt aber dieselbe Immobilie.
- Für den Kauf müssen wir uns dieses genauer ansehen – auf der Basis der beiden Attribute “datum” und “preis” kann man die Frage noch nicht gut beantworten.

Ist „Kauf“ ein Entity?



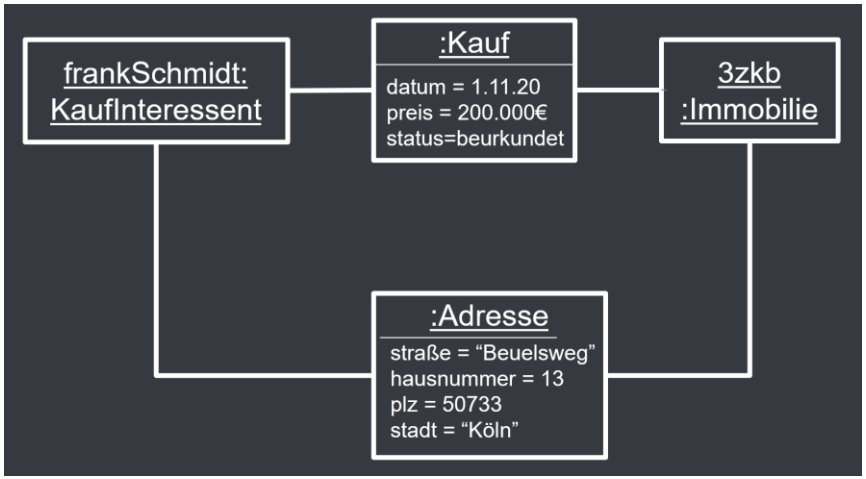
- Ein Immobilienverkauf ist kein *punktueller* Vorgang, sondern durchläuft verschiedene Stadien. Wir führen hierfür ein Statusfeld ein. (*Anmerkung*: die Statuswerte sind nicht groß recherchiert, sondern eher grob geschätzt – das reicht für unser Beispiel.)
- Damit erkennt man schon, dass sich der Status von “angezahlt” auf “vollständig bezahlt” ändern kann, und es bleibt doch derselbe Kauf(-vorgang).
- **Also ist Kauf auch ein Entity.**

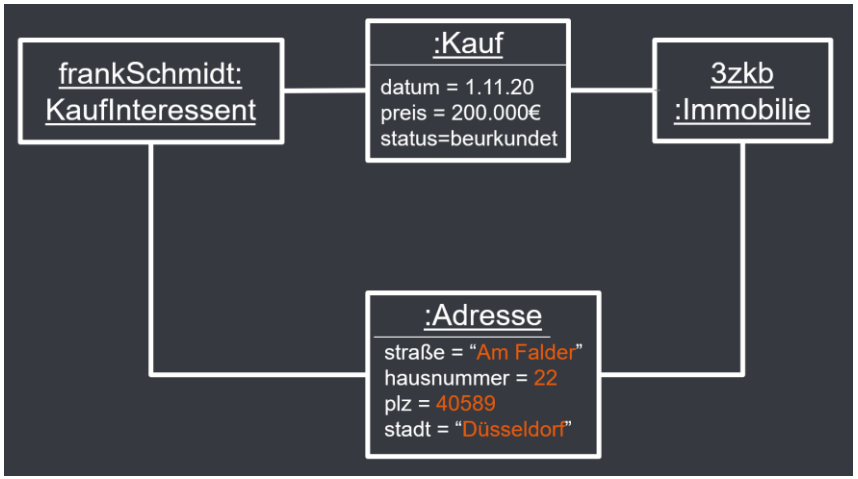
Ist Adresse ein Entity?



- Nehmen wir nun weiter an, dass wir für KaufInteressent und die Immobilie **Adressen** nachhalten wollen.
- Stellen wir dieselbe Frage für Adresse – auch ein Entity?

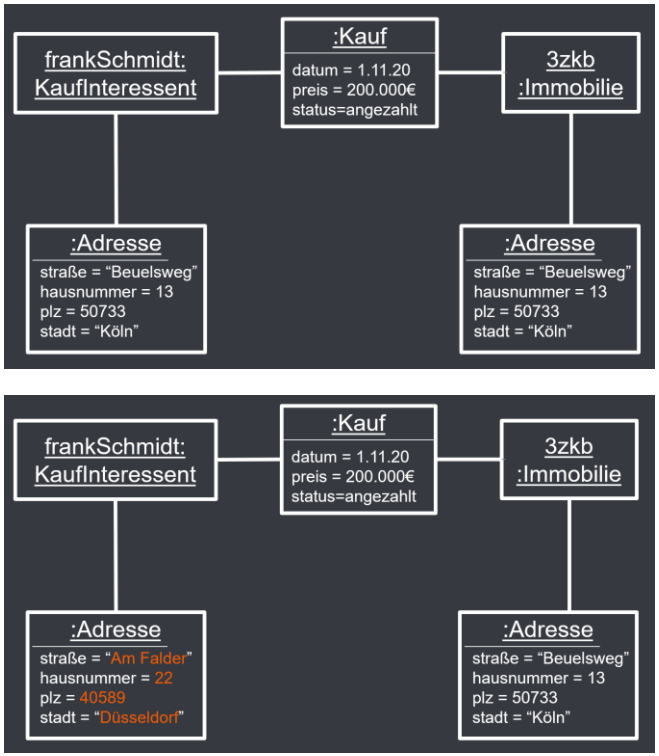
Ist Adresse ein Entity? => Objektdiagramm zur Klärung

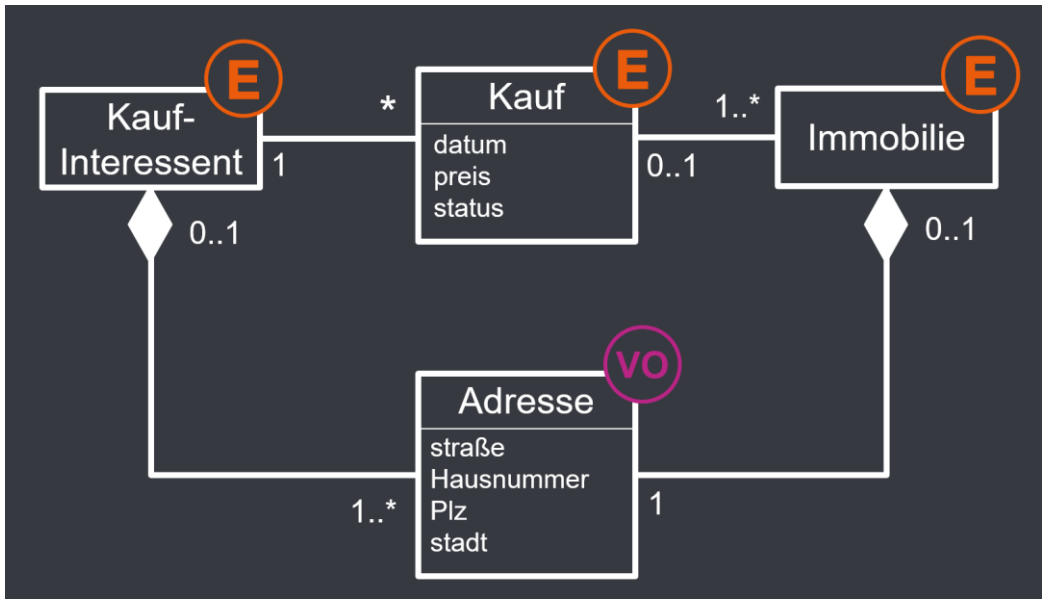




- Zur Klärung schauen wir uns ein *Objektdiagramm* an.
  - Im Objdiagramm werden (im Gegensatz zum Klassendiagramm) keine Klassen, sondern *Objektinstanzen* dargestellt. "*frankSchmidt : KaufInteressent*" ist also ein Objekt namens "frankSchmidt" von der Klasse "KaufInteressent".
- Hier sehen wir die Situation, dass Frank Schmidt in Köln am Beuelsweg 13 wohnt. Er will an derselben Adresse auch eine Immobilie kaufen – nämlich die Wohnung über der, in der er selbst wohnt.
  - Wenn Adresse ein Entity ist, dann müssten wir das so wie hier dargestellt modellieren können – Frank Schmidt verweist mit seiner Wohnadresse auf dasselbe Entity wie die zu kaufende Immobilie.
- Wenn Frank aber der Liebe wegen nach Düsseldorf umzieht (rechts), dann sieht man die Schwächen. Modelliert man so naiv wie oben dargestellt, dann würde sich auch die Adresse seiner zu kaufenden Immobilie ändern.

Besser: Adresse ohne Sharing





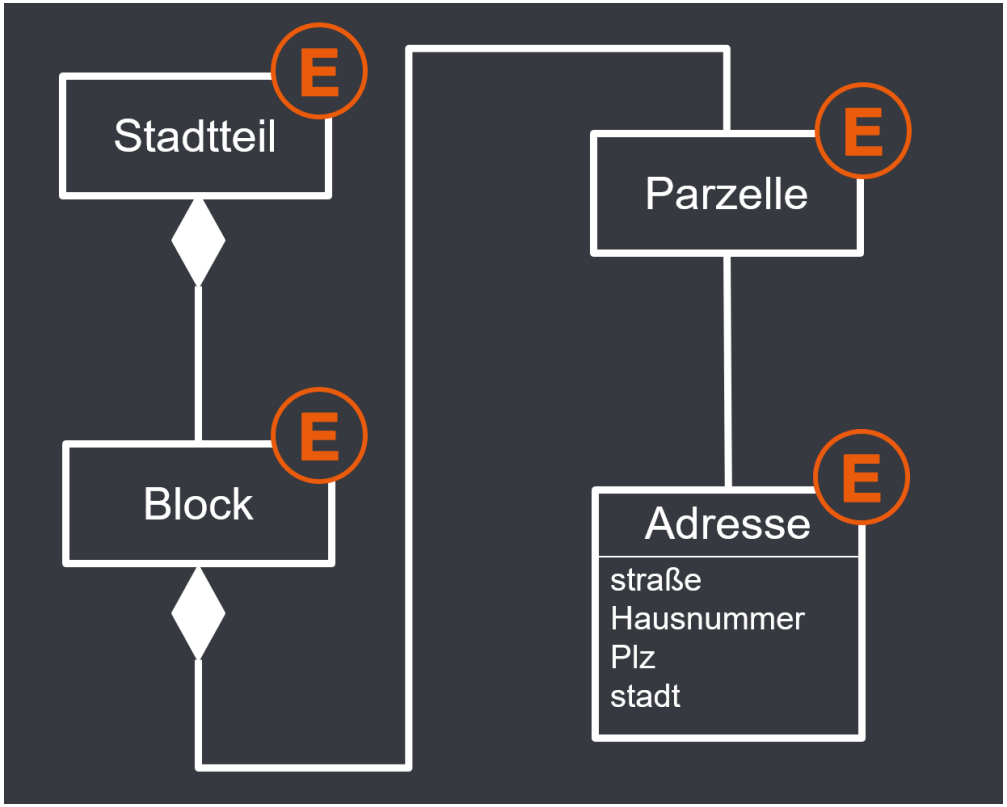
- Besser wäre es, die Modellierung so zu machen: Frank und die Immobilie haben von vornherein eigene Adressobjekte.
  - Dann mache ich keine Änderung an einem gemeinsam genutzten Adressobjekt. Vielmehr kann ich Franks alte Adresse "Beuelsweg in Köln" (links oben) wegwerfen und durch die neue Adresse "Am Falder in Düsseldorf" ersetzen (links unten).
- Damit wird Adresse zum sogenannten **Value Object** (rechts).

Value Objects

- Identität bestimmt durch Attributwerte
  - Andere Attributwerte => anderes Objekt
- Value Objects sind immutable
  - Attributwerte werden nur einmal gesetzt, und danach nicht mehr verändert.
  - Statt Änderung wird das Objekt gelöscht und neu erzeugt.
  - Damit vermeidet man Konfliktsituationen, die z.B. durch ein Sharing von Objektinstanzen entstehen, wie eben gesehen.

- Wie erkennt man Value Objekts? Vier Indizien:
  - ändert man nur einen Attributwert, wird es zu einem anderen Objekt
  - wird nicht mehr verändert, sondern eher gelöscht und neu angelegt
  - kein Sharing
  - stellt ein „komplexes Attribut“ eines anderen Geschäftsobjekts dar

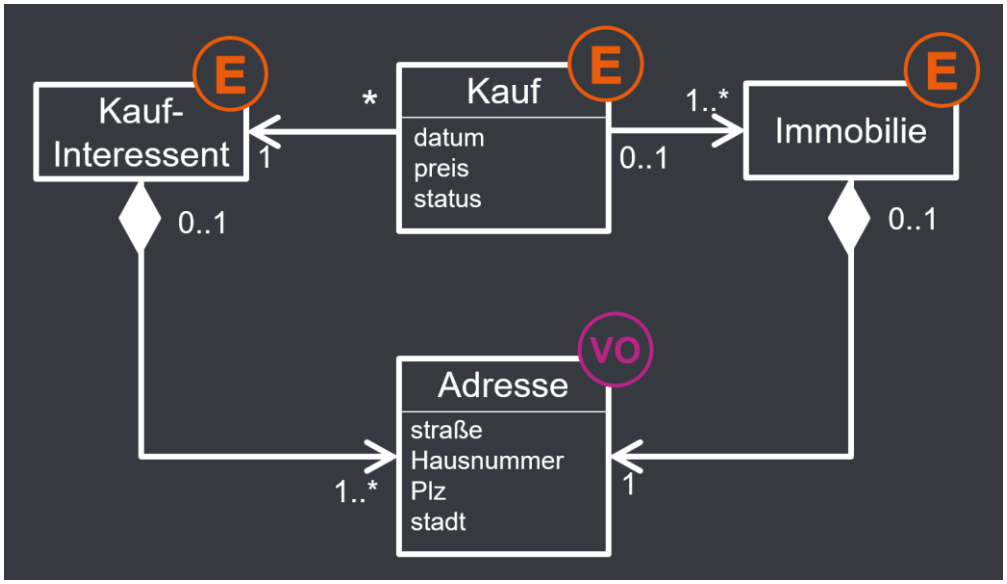
“Value Object oder Entity” ist kontextabhängig!



- In dem Maklerportal, im Gegensatz dazu, sind Änderungen an der Adresse **nicht im Fokus**. Man braucht die Daten, sie sind aber (anders als bei der Stadtplanung) nicht im Fokus des Softwaresystems.
- Genauso wie etwa eine Kontoverbindung ist sie ein komplexes Attribut, das man neu anlegt, wenn es sich ändert.

- Ob man eine Klasse als Entity oder Value Object modelliert, liegt nicht “in der Natur der Klasse”, sondern hängt vom Fokus und Kontext ab.
- Machen wir ein Gedankenexperiment als Beleg: nehmen wir an, wir würden kein Maklerportal, sondern eine **Stadtplanungs-Software** implementieren.
- Dann sähe unser Domain Modell vielleicht so aus links: Es gibt *Stadtteile*, die aus *Wohnblöcken* bestehen. Diese wiederum bestehen aus *Parzellen*. Eine Parzelle hat dann eine Adresse.
  - (Wahrscheinlich würde man die Adresse in dem Modell anders als links dargestellt modellieren, aber ich wollte die Adress-Klasse wiedererkennbar halten.)
- In diesem Beispiel würde man möglicherweise in einem zu planenen Neubaugebiet eine Straße noch einmal umbenennen. Oder man schneidet Parzellen neu zu und die Hausnummer ändert sich, weil es jetzt weniger oder mehr Parzellen in der Straße gibt.
- Hier wäre Adresse damit ein **Entity**!

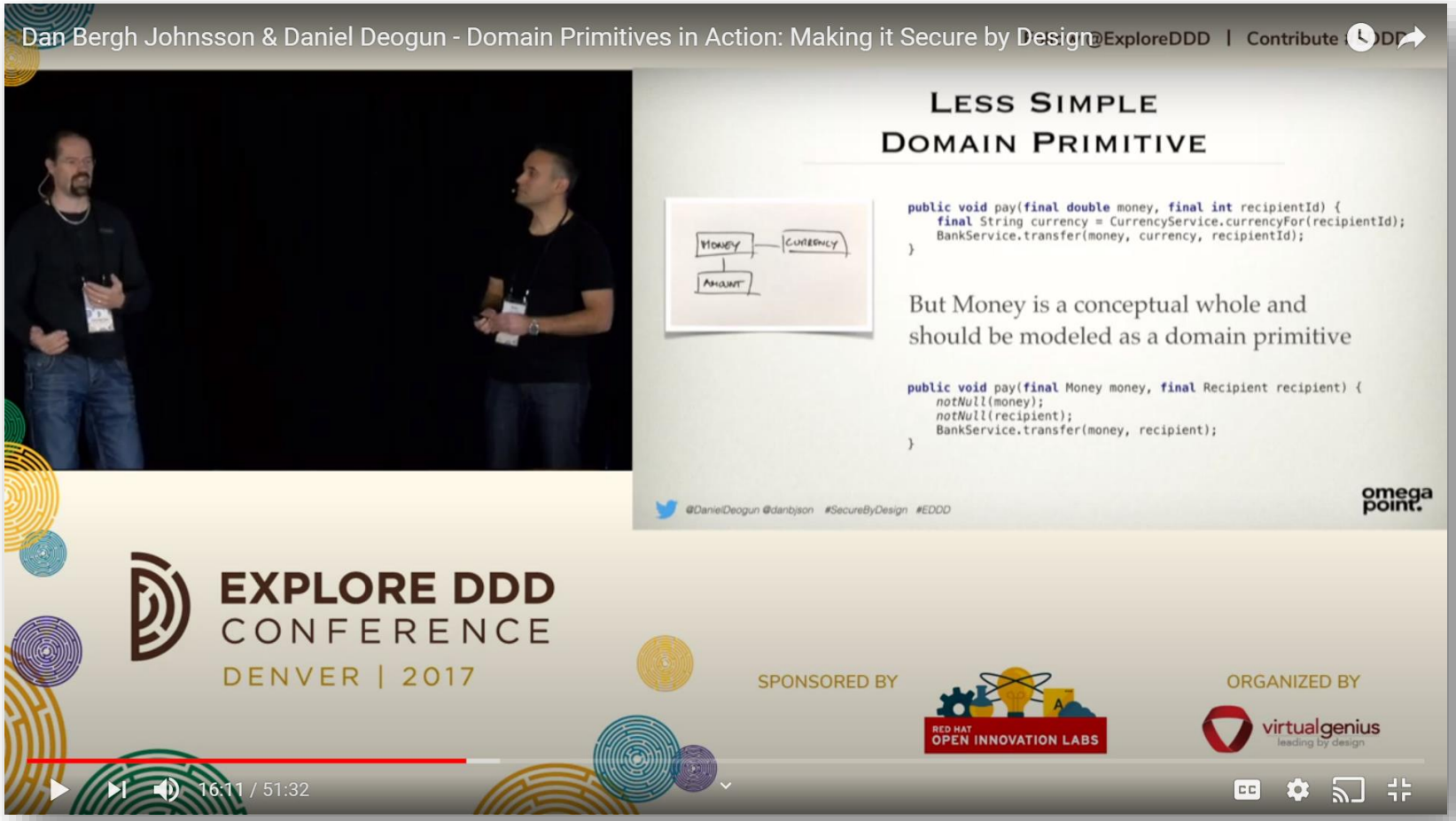
Und am Schluss ☺ - wofür braucht man das?



- Entities und Value Objects sind wichtige Konzepte aus dem **Domain-Driven Design (DDD)**, das uns noch länger beschäftigen wird.
- Sie sind für uns aus zwei Gründen wichtig:
  1. Beim Übergang von *ungerichteten* zu *unidirektionalen* Beziehungen haben wir hier Ausnahme von der sonst geltenden Regel “speziell => allgemein”. Es gilt immer die **Richtung “Entity => Value Object”**.
     - Da VOs keine eigene Identität haben, sondern sich nur über ihre Attributwerte definieren, können wir sie nicht per ID aus einer Datenhaltung holen. Dadurch wäre die Gegenrichtung nicht sinnvoll.
     - Außerdem haben wir VOs als eine Art von “komplexem Attribut” eingestuft. Attribute gehören eng zu einem Geschäftsobjekt, so dass ich vom GO direkt dorthin navigieren können will.
  2. Sobald wir zur Programmierung mit Spring Data JPA übergehen, sieht man, dass Entities und VOs unterschiedlich behandelt werden. Als Vorgriff: Entities haben eine ID und ein Repository, VOs nicht.

https://www.youtube.com/watch?v=DQrqVkgCDaM

Motivation für *Domain Primitives*



- Dan Bergh Johnsson & Daniel Deogun - Domain Primitives in Action: Making it Secure by Design
- https://www.youtube.com/watch?v=ogjOKlXHi08
- (12:40 – 16:10: Money Example)

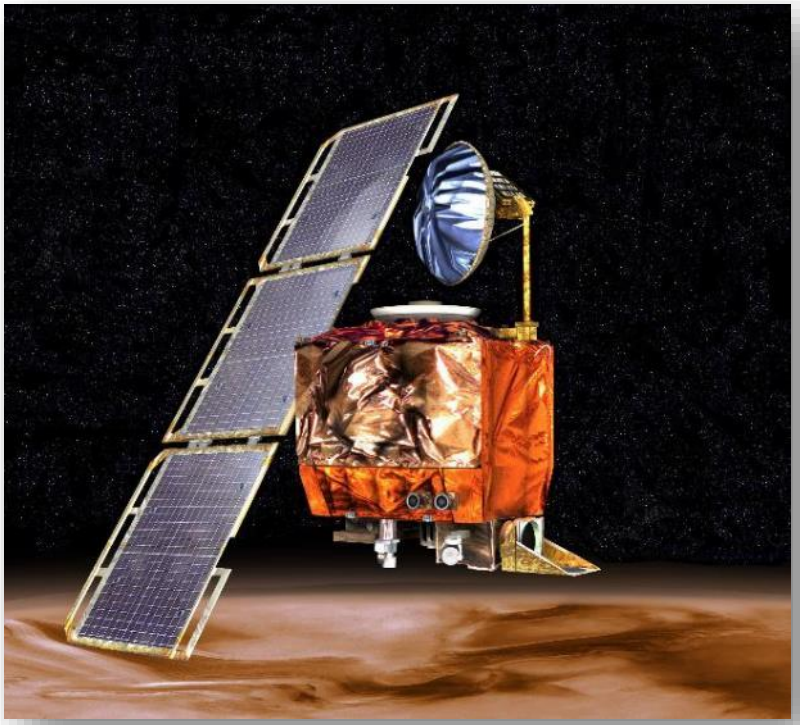
Zwei prominente (sehr teure) Fehler mit Basistypen

Ariadne-5-Absturz, 1996

“However, problems began to occur when the software attempted to stuff this 64-bit variable (…) into a 16-bit integer. (…) For the first few seconds of flight, the rocket’s acceleration was low, so the conversion between these two values was successful. However, as the rocket’s velocity increased, the 64-bit variable exceeded 65k, and became too large to fit in a 16-bit variable.“

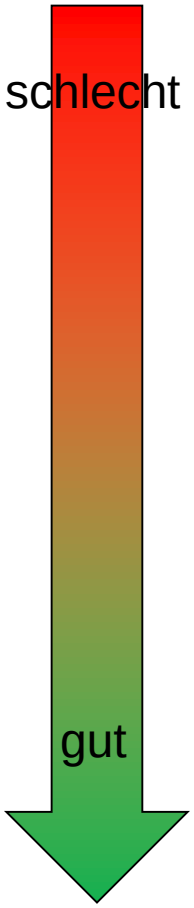
Absturz Mars Climate Orbiter, 1999

“Quality Assurance had not found the use of an imperial unit in external software, despite the fact that NASA’s coding standards at the time mandated use of metric units..“



Quellen: https://www.bugsnag.com/blog/bug-day-mars-climate-orbiter , https://www.bugsnag.com/blog/bug-day-ariane-5-disaster,
Bilder: NASA/JPL/Corby Waste - Gemeinfrei, https://commons.wikimedia.org/w/index.php?curid=390903,
DLR German Aerospace Center - Raumfrachter ATV-4 "Albert Einstein" Ariane 5ES Rollout_4, CC BY 2.0, https://commons.wikimedia.org/w/index.php?curid=26533952

Die Rangfolge von „am schlechtesten“ bis „am besten“



1. ~~Attribute als primitive Datentypen (int, float, double, …)~~

Sollte man immer vermeiden (ist ja auch leicht umzusetzen)

2. Attribute als Wrapperklassen (Integer, Float, Double, …)

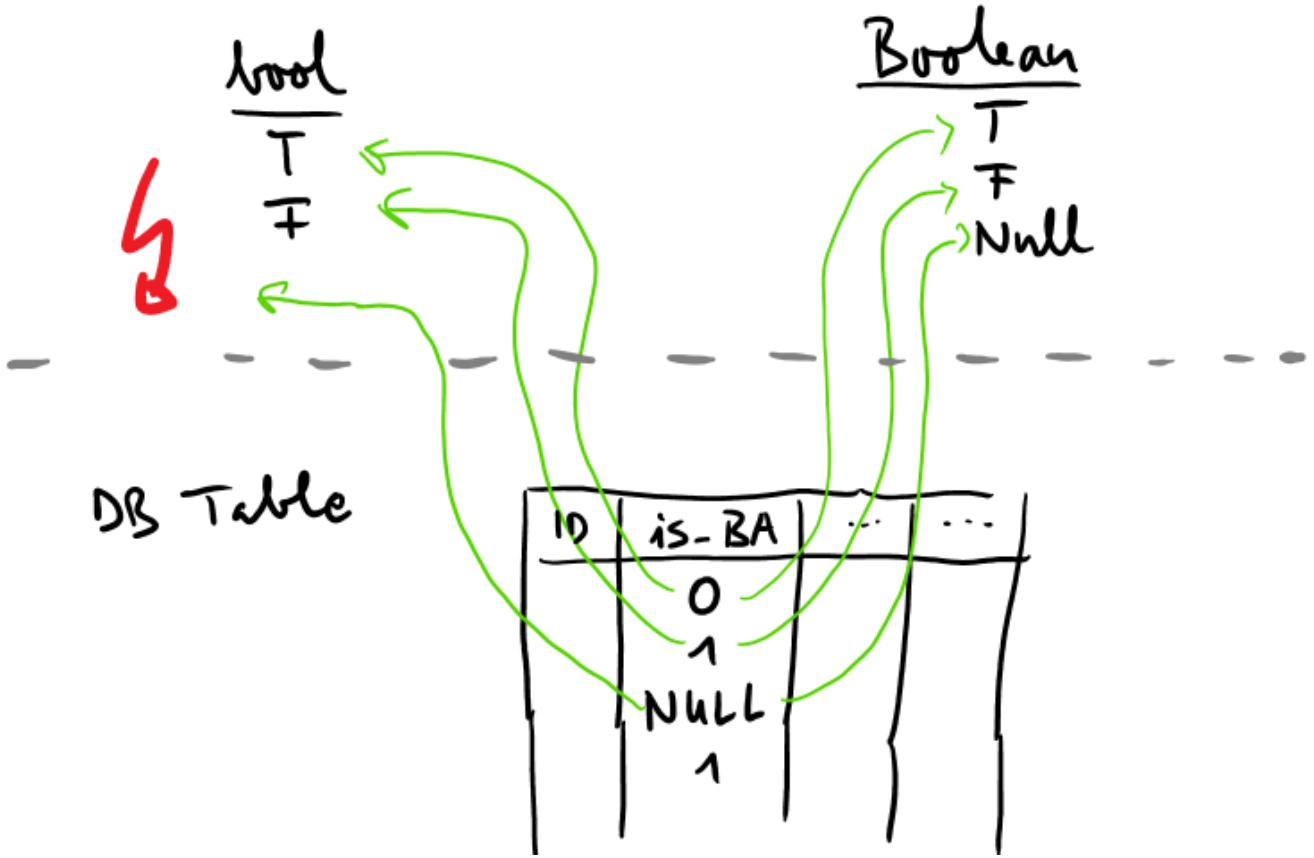
3. Attribute als Wrapperklassen mit domänen-spezifischer Validierung

4. Attribute als „Domain Primitive“ Value Objects
   - Erlaubt Builder / Konverter, spezifische Validations / Exceptions, domänenspezische Funktionalität

5. ~~Attribute als native *Domain Primitives*~~
   - Voller IDE-Support, so wie native Datentypen, aber domänen-spezifisches Verhalten

zumindest in Java leider nicht ver-fügbar, ggfs. in moderneren Sprachen

Option 1) Warum keine „primitive data types“ in Entities?



- Bei einer NULLABLE DB Column kann eine NULL nicht auf einen primitiven Datentypen abgebildet werden, auf einen Wrapper-Typen aber schon (siehe Beispiel).
- Generell sind Frameworks wie Spring eher auf den Umgang mit Objekten ausgelegt; die Wrapperklassen sind damit konsistenter zu benutzen.

Option 3) Wrapperklasse + JPA-Validierung

Beispiel aus dem Campus-Management-System *(kam im Video nicht vor)*

Student

```
@DecimalMin(value = "10000000")
@DecimalMax(value = "99999999")
private Long matrNum;
```

Test

```
@Test
public void testValidation() {
   Student s = new Student();
   s.setName("Hans");
   s.setMatrNum(1234567L);
   Exception ex = assertThrows(TransactionException.class, () -> {
      studentRepository.save(s);
   });
   s.setMatrNum(10000000L);
}
```

- Unsere Domänenregel sagt:
  „Eine Matrikelnummer beginnt mit 1-9 und ist genau 8-stellig“ (*)

- Wertebereich von Attributen kann durch Annotationen wie `@DecimalMin(value = "10000000")` eingeschränkt werden. Es gibt auch noch weitere Annotationen zur Validierung. Damit kann zumindest verhindert werden, dass Entities mit Werten erzeugt werden, die in der Domäne keinen Sinn ergeben.

(*) Ich habe nicht recherchiert, ob diese Regel für die TH Köln genau so stimmt – nehmen wir es einfach mal an.

Definition „Domain Primitive“

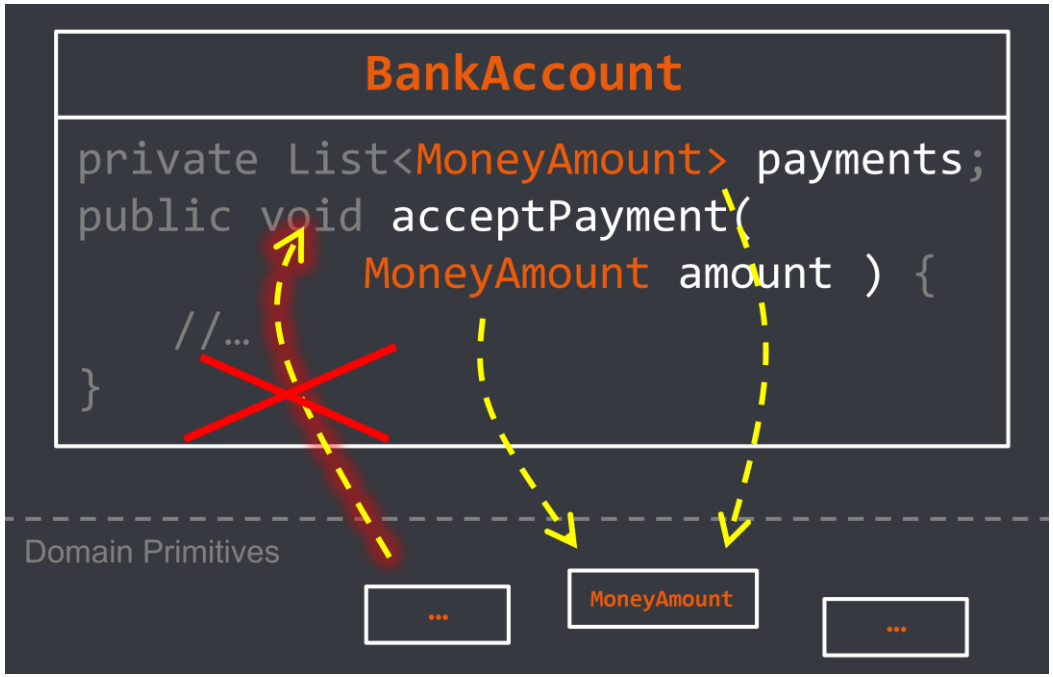
- A value object precise enough in its definition that it, by its mere existence, manifests its validity is called a **domain primitive**.
  - *Robert Clark (2019), https://medium.com/@robot88/domain-primitives-swift-and-you-cb9752c44878*
- […] we require invariants to exist and they must be enforced at the point of creation.
  - *D. B. Johnsson, D. Deogun, D. Sawano (2018), https://freecontent.manning.com/domain-primitives-what-they-are-and-how-you-can-use-them-to-make-more-secure-software/ [Johnsson, Deogun, Sawano]*

- Eigenschaften von Domain Primitives *[Johnsson, Deogun, Sawano]*
  - Their invariants are checked at the time of creation.
  - They can only exist if they’re valid.
  - They should always be used instead of language primitives or generic types.
  - Their meaning’s defined within the boundaries of the current domain, even if the same term exists outside of the current domain.

Regeln für Domain Primitives

1. Value Object
   - keine Setter
   - Änderungen geben neues Objekt zurück (immutable)
2. Erzeugung über Factory-Methode
   - protected oder private Constructor
   - valueOf(…), fromXXX(…)
     - Damit kann man die Semantik der Factory-Methoden besser deutlich machen als über Konstruktoren.
3. Exceptions für Fehler bei Erzeugung
4. Keine Rückreferenzen auf Domain-Entities

Wieso keine Rückreferenzen auf Domain-Entities?



- Eine Referenz kann durch eine Member-Variable eines Typs entstehen, aber auch schon dadurch, dass eine Methode einen Parameter eines Typs hat.
  - Im obigen Beispiel: die Domain Primitive „ MoneyAmount“ kommt im Entity „BankAccount“ 2x vor, einmal als Member, einmal als Parameter.
- Das ist auch völlig ok: Nach dem Depency Inversion Principle soll ich in meinem Code keine volatilen (sich oft ändernden) Konzepte referenzieren. Aber Domain Primitives bilden „Basiskonzepte“ meiner Domäne ab, die werden sich nicht oft ändern. (Oder denken Sie, dass sich der Begriff und die Regeln für „Geldbetrag“ oft ändern?)
- Nur eine Rückreferenz darf auf keinen Fall sein: Dann nämlich ist meine Domain Primitive plötzlich von „BankAccount“ abhängig. Und dann kann ich nicht mehr einfach überall in meinem „MoneyAmount“ benutzen, ohne vorher drüber nachzudenken.

Beispiel für Domain Primitives (1): **Strength / Impact**

Text Adventure *(dieses Beispiel wurde im Video gezeigt)*

Strength (Teil 1)

```
public class Strength {
   private Float max;
   private Float current;


   public static Strength fromMax( Float maxStrength ) { return new Strength( maxStrength ); }

   public static Strength fromCurrentAndMax( Float current, Float maxStrength ) {
      Strength strength = new Strength( maxStrength );
      strength.setCurrent( current );
      return strength;
   }

   public Strength afterImpact( Impact impact ) {
      if ( impact.isNone() ) {
         return this;
      }
      else {
         float newCurrent = this.current + impact.effectOnStrength();
         if ( newCurrent > max ) newCurrent = max;
         if ( newCurrent <= 0f ) newCurrent = 0f;
         return Strength.fromCurrentAndMax( newCurrent, max );
      }
   }

   public Float relative() { return current / max;  }
   public boolean meansDeath() { return current <= 0f;  }
   //...
```

- In dem Beispiel sieht man gut die typischen Eigenschaften einer Domain Primitive:

1. Der „Kontext“ der maximalen Stärke wird zusammen mit der aktuellen Stärke gespeichert. Dadurch kann man immer eine relative Stärke ausrechnen (z.B. für farbliche Darstellung).
2. Es ist ein Value Object, es gibt keine Setter.
3. Es gibt verschiedene Factory-Methoden: üblicherweise wird eine Strength nur über die maximale Stärke initialisiert, aber es gibt auch eine Methode, um eine „geschwächte Stärke“ (wo current < max) zu erzeugen, z.B. zu Testzwecken.
4. „afterImpact“ nimmt einen Impact und wendet ihn auf die Strength an. Als Ergebnis wird eine neue Strength zurückgegeben (Value Object, Immutable)

Beispiel für Domain Primitives (1): **Strength / Impact**

Strength (Teil 2)

```
public class Strength {
  ...

  @Override
  public String toString() {
    return String.format("%.01f", current) + " out of " + String.format("%.01f", max);
  }

  @Override
  public boolean equals(Object o) {
    if (this == o) return true;
    if (o == null || getClass() != o.getClass()) return false;
    Strength strength = (Strength) o;
    return Float.compare(strength.max, max) == 0 &&
           Float.compare(strength.current, current) == 0;
  }

  protected Strength( Float maxStrength ) {
    if ( maxStrength <= 0f ) throw new StrengthException( "Maximum strength must be > 0" );
    this.max = maxStrength;
    current = maxStrength;
  }

  protected void setCurrent( Float current ) {
    if ( current < 0f ) throw new StrengthException( "Current strength must be >= 0" );
    if ( current > this.max ) throw new StrengthException( "Current strength must be < max" );
    this.current = current;
  }
}
```

5. Man kann toString und equals sinnvoll überschreiben.
6. Der „protected“ Konstruktor wirft eine spezielle StrengthException, wenn man versucht, eine maximale Stärke <=0 zu initialisieren.

Beispiel für Domain Primitives (1): **Strength / Impact**

Impact

```
public class Impact {
   private Float effectOnStrength;

   protected Impact( Float effectOnStrength ) {
      this.effectOnStrength = effectOnStrength;
   }

   public static Impact from( Float effectOnStrength ) {
      return new Impact( effectOnStrength );
   }

   public boolean isNone() {
      return Float.compare(effectOnStrength, 0f) == 0;
   }

   /**
    * Visibility "package" => only to be used by Strength.receiveImpact( impact )
    * @return changeToStrength
    */
   Float effectOnStrength() {
      return effectOnStrength;
   }

   @Override
   public String toString() {
      return String.format("%.01f", effectOnStrength);
   }
}
```

- Auch Impact wird konsequent als Value Object (nicht als Float) implementiert.
- Damit hält man sich z.B. für später offen, mal eine andere Art von Impact zu haben – nicht nur so etwas wie „-10“, sondern auch „halbiere die Stärke“. So eine Zusatzinformation kann man in dem Value Object gut unterbringen.

Beispiel für Domain Primitives (2): **MatrNumber**

Beispiel aus dem Campus-Management-System *(kam im Video nicht vor)*

Student

```
@Getter
@Setter
//@DecimalMin(value = "10000000")
//@DecimalMax(value = "99999999")
//private Long matrNum;
@Embedded
private MatrNumber matrNum;
```

StudentRepository

```
//List<Student> findByMatrNumLessThan( Long matrNum );
List<Student> findByMatrNumLessThan( MatrNumber matrNum );
```

CreateSampleData

```
//s1.setMatrNum( 12345678L );
s1.setMatrNum( MatrNumber.fromLong( 12345678L ) );
```

- Die bessere (aber auch aufwändigere) und „DDD-konformere“ Lösung ist es, *nur* Value Objects als Attribute zuzulassen. Das hat den Vorteil, dass ich neben einer maßgeschneiderten Validierung auch Builder-Methoden und sonstige Domänen-spezifische Logik unterbringen kann.
- **Achtung:** Dieses Vorgehen ergibt nur dann Sinn, wenn man systematisch eine Bibliothek von „Domänenspezifischen atomaren Value Objects“ aufbaut und diese konsequent wiederverwendet. Für jedes Entity neue Value Objects einfach um Integer, String etc. „herum zu wrappen“ ist nutzlos und kontraproduktiv (nur Aufwand, kein Nutzen).

Beispiel für Domain Primitives (2): **MatrNumber**

Beispiel aus dem Campus-Management-System *(kam im Video nicht vor)*

MatrNumber

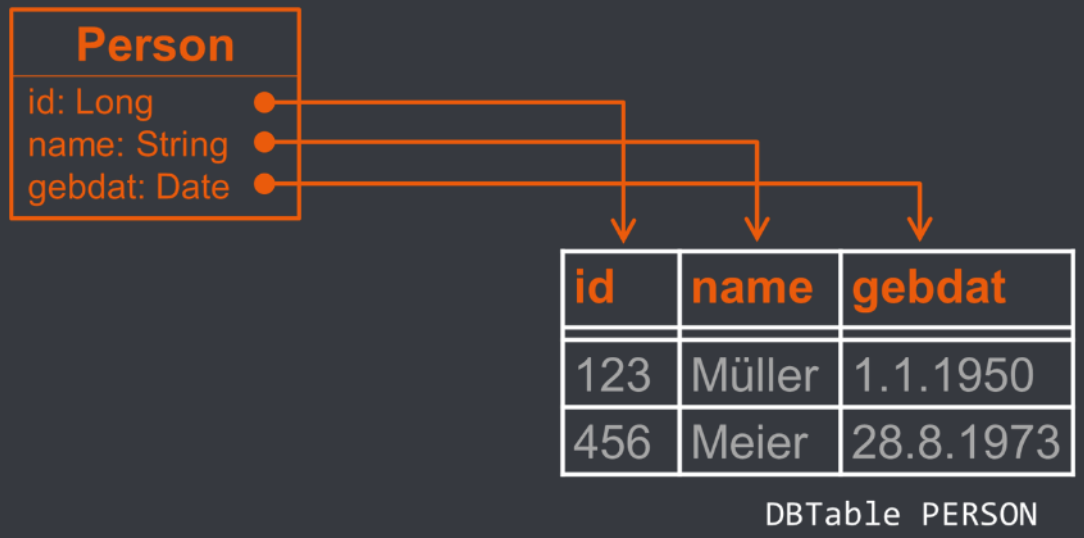
```
package thkoeln.st2.domainprimitives;
import …

@ToString
@EqualsAndHashCode
@Embeddable
public class MatrNumber {

   @DecimalMin(value = "10000000")
   @DecimalMax(value = "99999999")
   Long num;

   private MatrNumber( Long l ) {
      if (l > 99999999L || l < 1000000L ) {
         throw new MatrNumberException( "invalid number" + l );
      }
      num = l;
   }
   protected MatrNumber() {}

   // a number of builder methods
   public static MatrNumber fromLong( Long l ) {
      MatrNumber m = new MatrNumber( l );
      return m;
   }
   public static MatrNumber fromString( String s ) {
      throw new UnsupportedOperationException( "not implemented yet");
   }
   public static MatrNumber createUniqueNew() {
      throw new UnsupportedOperationException( "not implemented yet");
   }
}
```

MatrNumberException

```
package thkoeln.st2.domainprimitives;

public class MatrNumberException extends
RuntimeException {
   public MatrNumberException( String msg ) {
      super( msg );
   }
}
```

ORM und Repositories

Wie speichert man nach DDD ein Entity in einer Datenbank?



Technology
Arts Sciences
TH Köln

https://www.youtube.com/watch?v=vmzGGslfyXg&t=49s

Lebenszyklus eines Geschäftsobjekts (Entity / Aggregate)

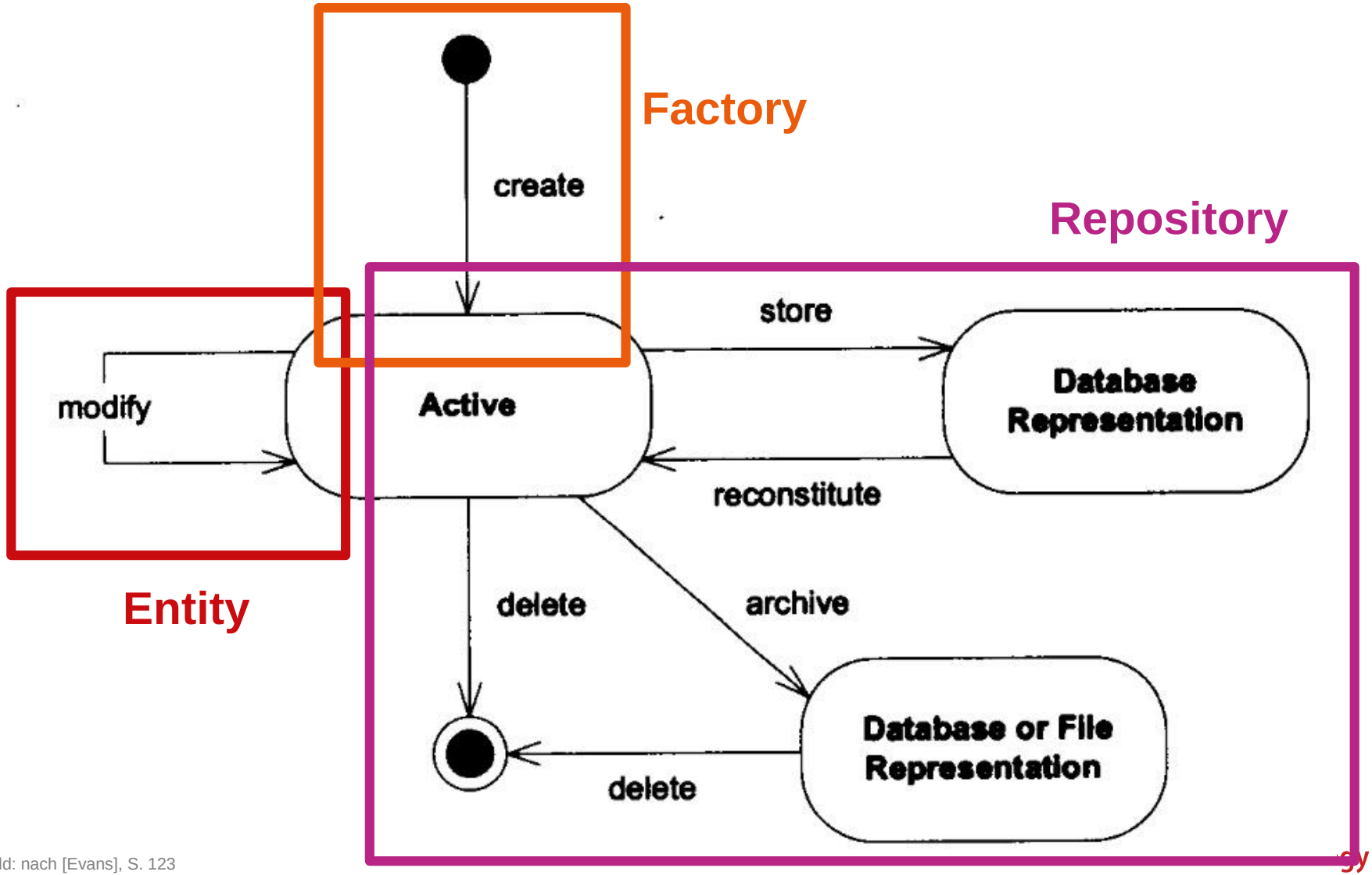


Bild: nach [Evans], S. 123

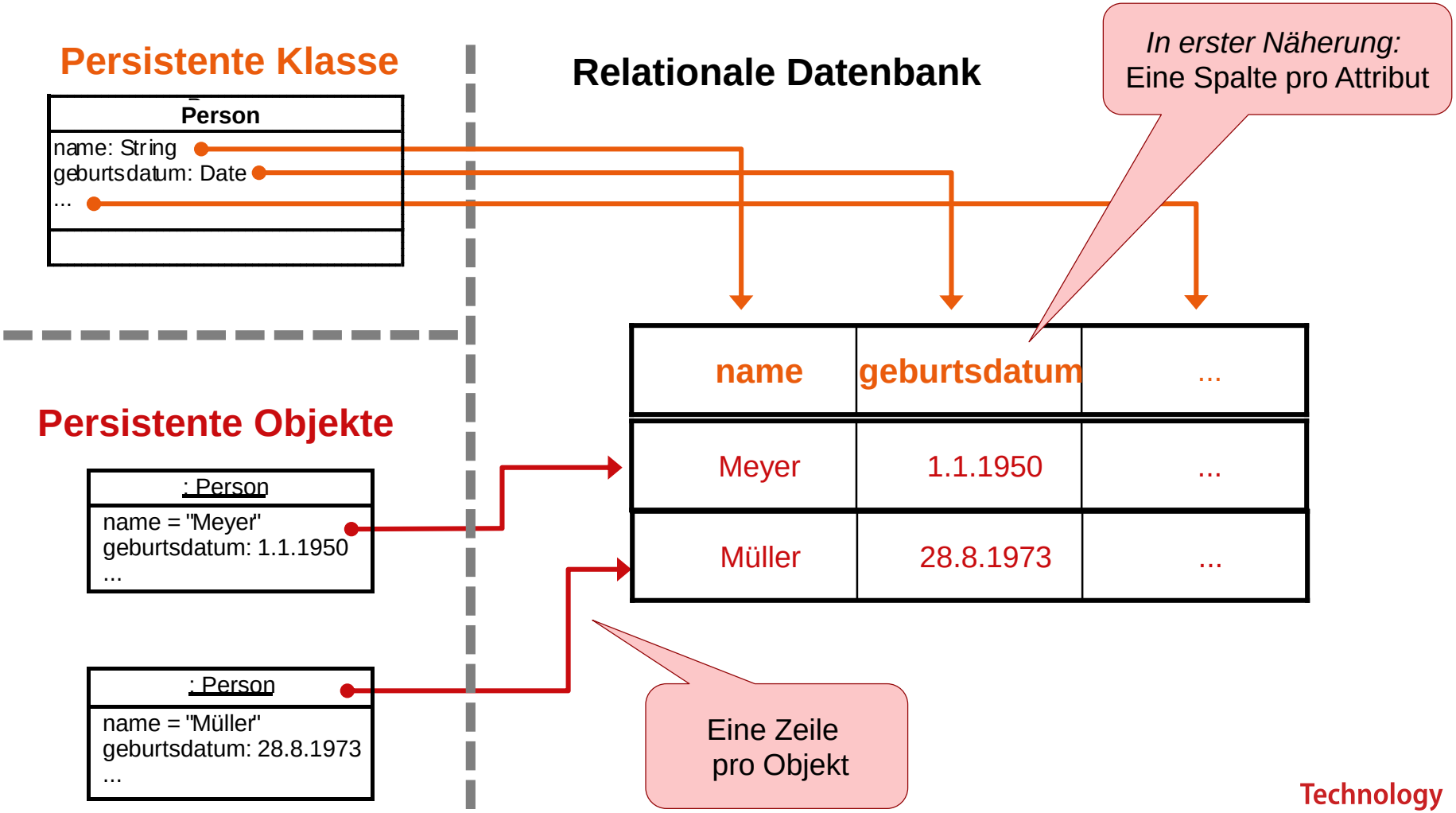
Warum Datenbanken für persistente Datenspeicherung?

Warum keine Dateien?

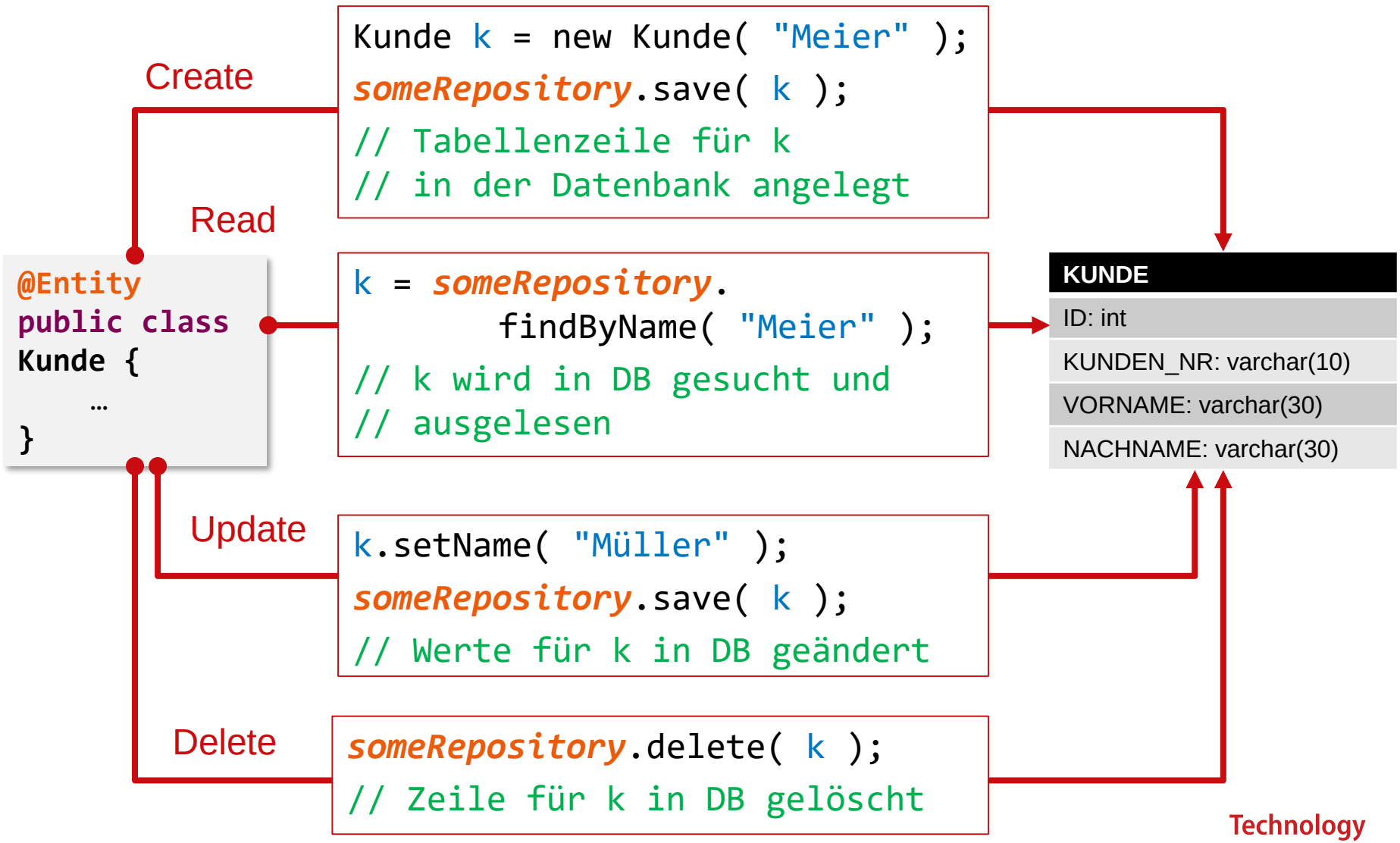
- Redundanzfreiheit
  - Daten/Objekte werden nur einmal gespeichert
- Konsistenz
  - Verschiedene Anwendungen greifen auf eine zentrale Datenhaltung zu
- Zuverlässigkeit
  - Datenbanksystem ist optimiert für Robustheit und Recovery
- Abfragen
  - Komplexe Abfragen durch Abfragesprachen unterstützt (z.B. SQL)
- Rollen- / Rechte-Konzept
  - Zugriffsrechte für Teile der Daten feingranular steuerbar
- Mehrbenutzerbetrieb
  - Paralleler Zugriff mehrere Benutzer/Anwendungen möglich
- Transaktionskonzept
  - Rollback bei Fehlschlägen problemlos
- **Wir konzentrieren uns auf relationale DBMS**
  - … diese stellen den häufigsten Fall in heutigen IT-Anwendungen dar

Quelle: nach [Winter]

ORM = Object-Relational Mapping
= Abbildung von Klassen auf Tabellen



"Heutiger" Anspruch an ORM (als Pseudocode)



Repository: Speichern, Suchen und Löschen

Pseudocode, konzeptionelle Sicht

```
public interface CarRepository
{
   // add, update, or delete car from database
   void save( Car car );
   void delete( Car car );

   // find one specific car, or all cars in database
   Car findById( ID id );
   List<Car> findAll();

   // query cars according to attribute values
   List<Car> findByBrand( String brand );
   List<Car> findByMileageGreaterThan( Double mileage );
}
```

Spring Data JPA in Action

Object-Relational Mapping (ORM) für Entities mittels Java Persistence API (JPA) und Spring

```
@Entity
public class Car
{
  // ...
}
```

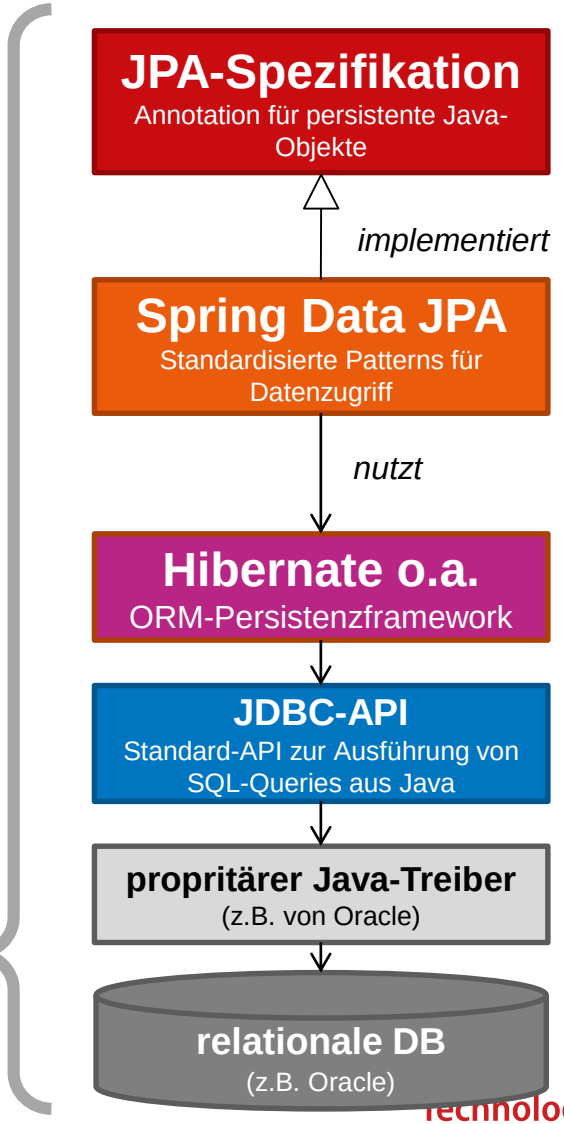
Technology
Arts Sciences
TH Köln

https://www.youtube.com/watch?v=iCo4Gj_h5xQ

JPA = Persistente Java-Objekte durch Annotation

- JPA 1.0: 2006 - aktuell: JPA 2.1 (2013)
  - wird als Jakarta Persistence weitergeführt (2.2)
  - Mit JPA sollten wurde ein Standard-Layer für Persistenzframeworks wie Hibernate und TopLink definiert
- In dieser Vorlesung: Spring Data JPA
  - Implementiert JPA
  - Nutzt Persistenzframework wie Hibernate
- => **Standardisiertes Design der Persistenzschicht**

Technologie-Stack (vereinfacht)

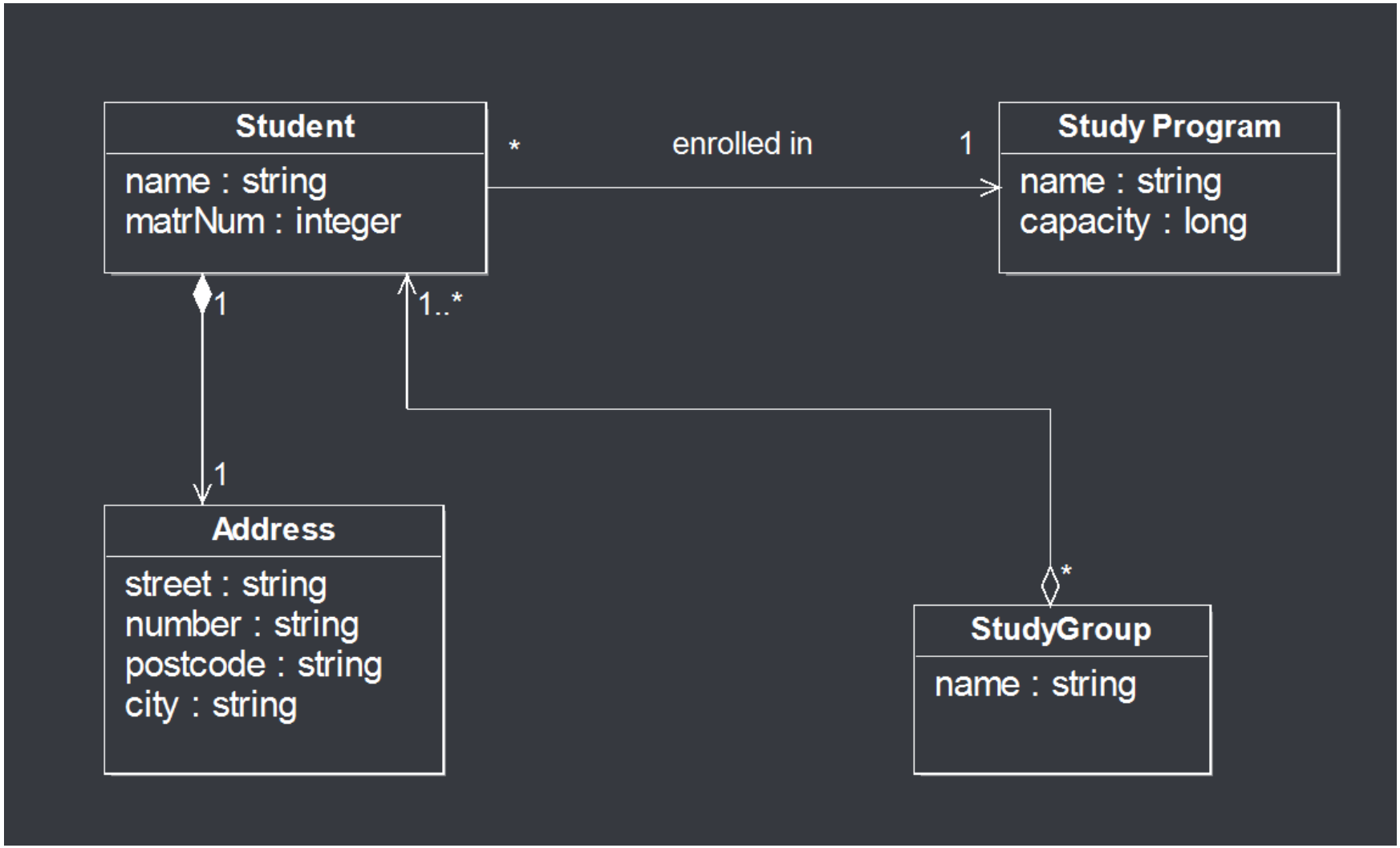


Hintergründe zu JPA

- Der Fokus der Vorlesung ist auf Umsetzung von DDD
  - Spring Data JPA wird als praktisches Hilfsmittel gesehen
  - => keine detaillierte Einführung, Fokus auf wesentliche JPA-Tags
- JPA (im späteren Software-Alltag …) kann komplex sein
  - Es geht nichts über selbst ausprobieren! Das müssen Sie im Praktikum dann auch tun.
- DDD-Regeln helfen, unnötige technische Komplexität zu vermeiden!
  - *Beispiel:* bidirektionale n:m-Beziehungen gehen in Spring JPA, sind aber „tricky“. DDD rät von bidirektionalen Beziehungen ab => **siehe auch Video dazu, im Rahmen des Logischen Datenmodells.**

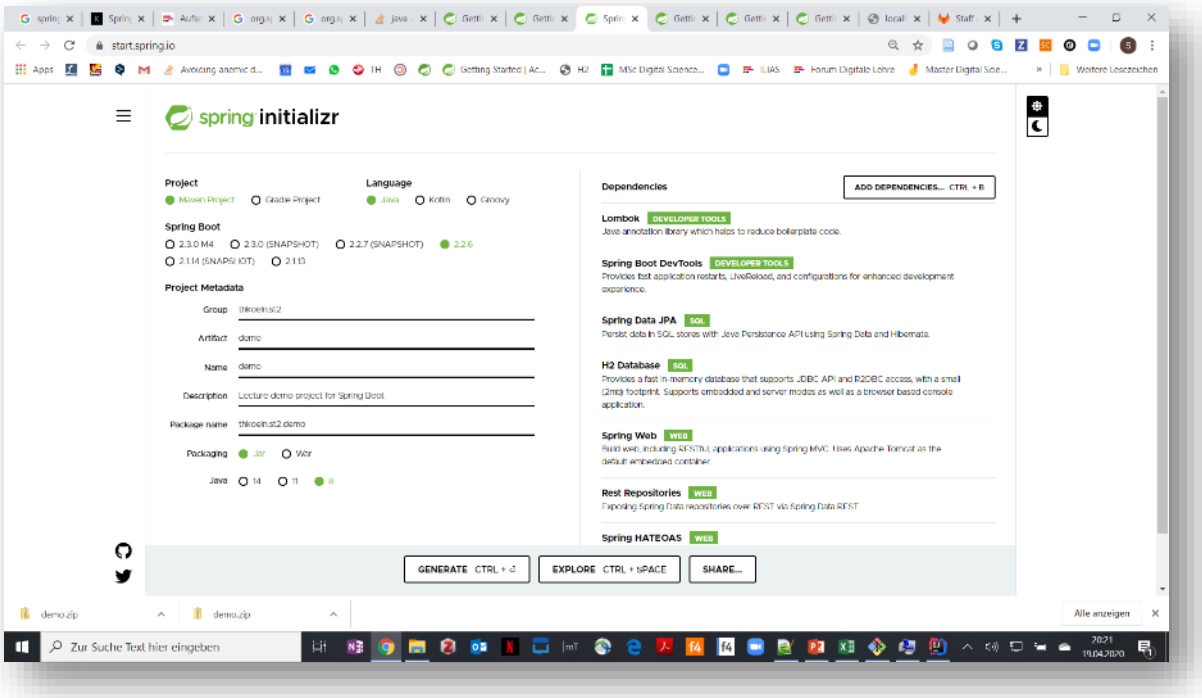
- Weiterführende Quellen:
  1. Keith, Mike, and Merrick Schincariol. *Pro JPA 2 (Expert's Voice in Java) 2nd Edition,* Apress, 2013. https://link.springer.com/book/10.1007%2F978-1-4302-4927-6 *(für Studierende der TH Köln zugänglich)*
  2. Spring Data JPA Referenz: https://docs.spring.io/spring-data/jpa/docs/current/reference/html/
  3. JPA Cheat Sheet (© Philipp Meier, CC-BY-2.0): (in Klausur verfügbar!) http://www.cheat-sheets.org/saved-copy/javaEE6ReferenceSheet.pdf (zweite Seite ist relevant)

Code-Beispiel



Wie probiere ich am besten selbst aus?

- **spring initializr**: https://start.spring.io/



- **Achtung:** So erstellt man sich ein Projekt zum Ausprobieren.
- **Das ist NICHT der Weg für den Meilenstein M1** im ST2-Praktikum!

Ein einfaches Entity

Entity

```
package thkoeln.st2.demo.entities;

import lombok.Getter;
import lombok.Setter;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;

@Entity
public class Student {
   @Id
   @GeneratedValue(strategy = GenerationType.AUTO)
   @Getter
   private Integer id;

   @Getter
   @Setter
   private String name;

   @Getter
   @Setter
   private Long matrNum;

   public Student() {}
}
```

Repository

```
package thkoeln.st2.demo.repositories;

import org.springframework.data.repository.CrudRepository;
import thkoeln.st2.demo.entities.Student;

import java.util.List;

public interface StudentRepository extends
                           CrudRepository<Student, Long> {
   Student findByName( String name );
   List<Student> findByMatrNumLessThan( Long matrNum );
}
```

Sampledaten und Tests

Sample-Daten-Erzeugung zum Startup

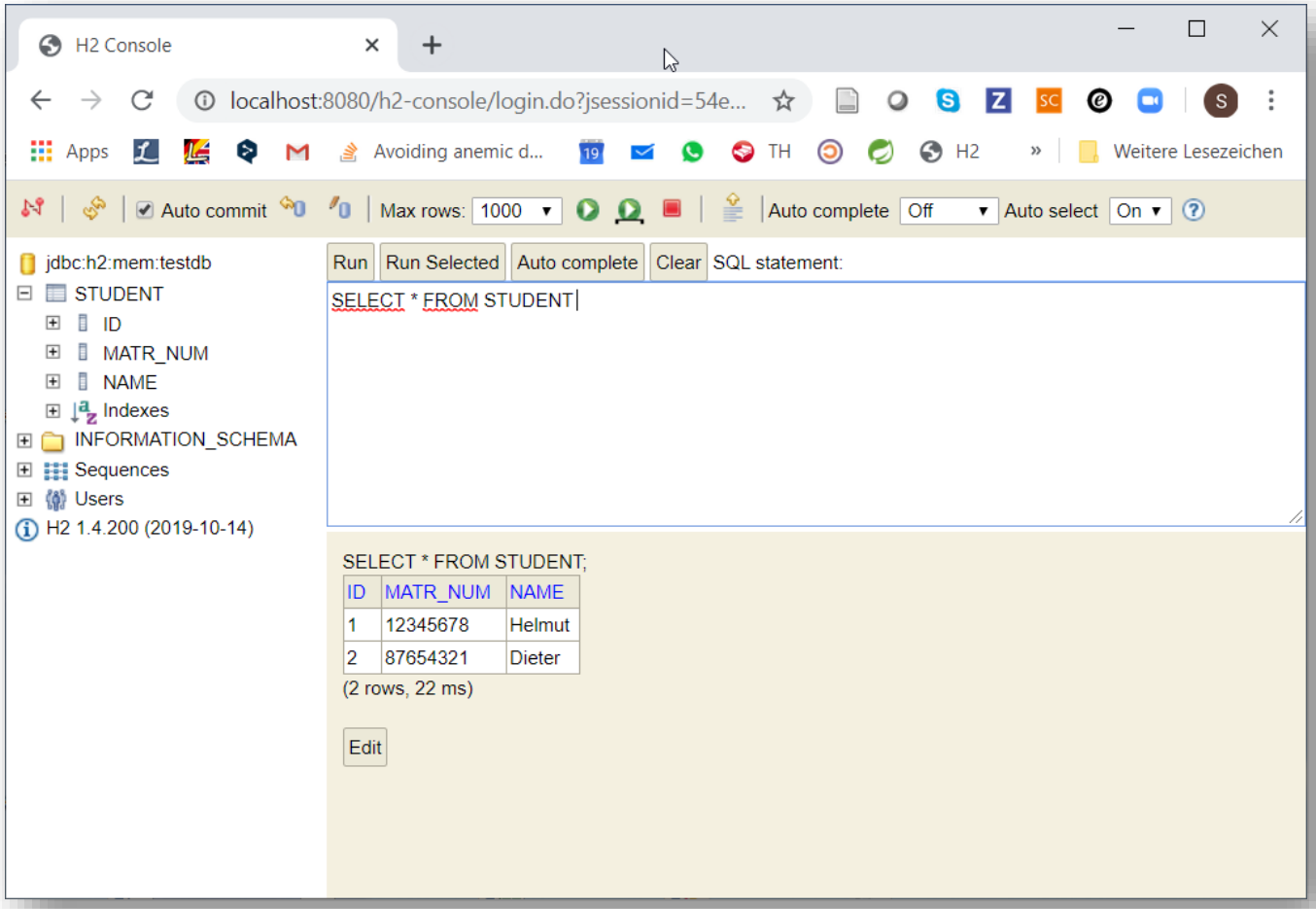
```
package thkoeln.st2.demo;

import ...

@Component
@Transactional
public class CreateSampleData implements
             ApplicationListener<ContextRefreshedEvent> {

   @Autowired
   private StudentRepository studentRepository;

   @Override
   @Transactional
   public void onApplicationEvent(ContextRefreshedEvent arg0) {
      // wesentliche Entitäten anlegen
      createStudents();
   }

   private void createStudents() {
      // ...
   }
}
```

Unit-Test

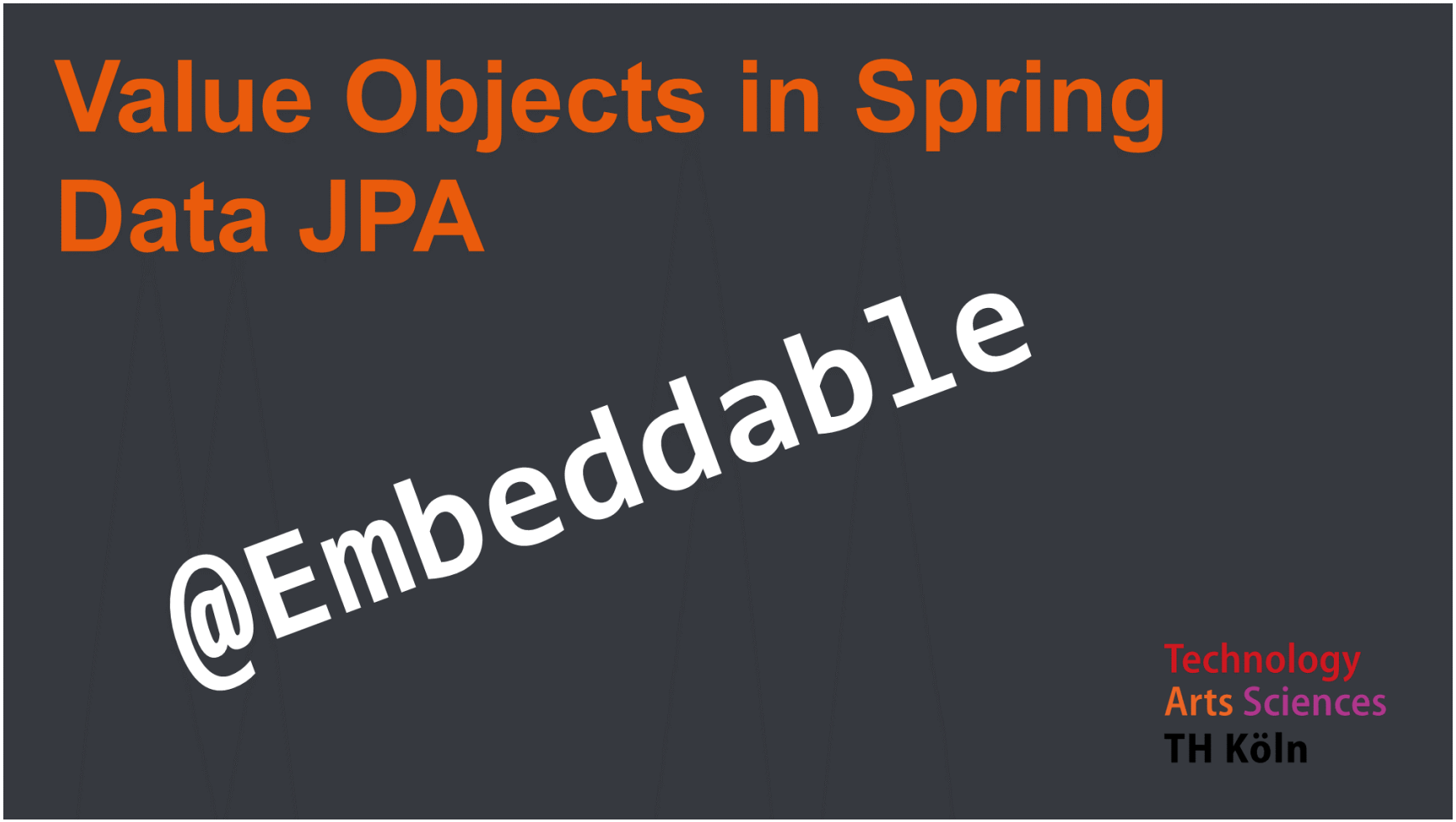
```
package thkoeln.st2.demo;

import ...

@SpringBootTest
class DemoApplicationTests {

   @Autowired
   private StudentRepository studentRespository;

   @Test
   public void fetchStudents() {
      // test finder Method
      List<Student> list =
             studentRespository.
             findByMatrNumLessThan( 51234567L );
      assert( list.size() == 1 );
      assert( list.get( 0 ).getName().
              equals( "Helmut") );
   }
}
```

H2 Console zur Prüfung der Datenbank

- H2 Console: localhost:8080/h2-console/



Value Objects in Spring Data JPA



https://www.youtube.com/watch?v=BGyfj2ixPy8

Adresse ergibt als Entity keinen Sinn …

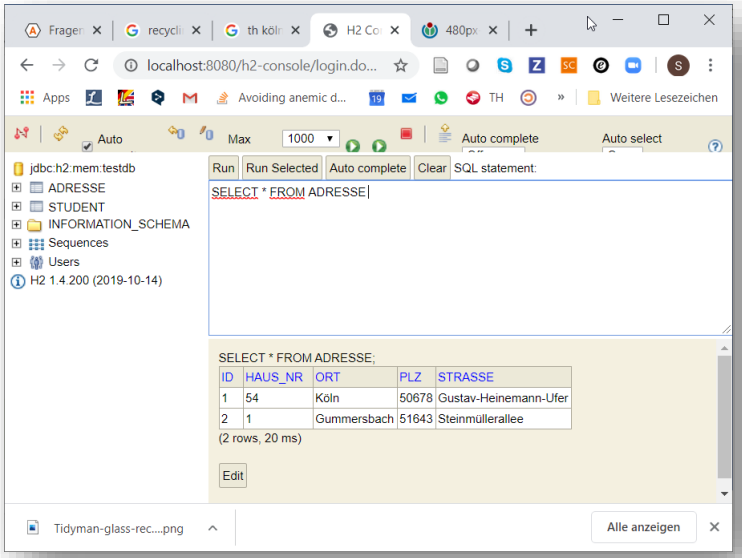
Adresse als Entity

```
@Entity
public class Adresse {

   @Id
   @GeneratedValue(strategy =
             GenerationType.AUTO)
   private long id;
   private String plz;
   private String ort;
   private String strasse;
   private Integer hausNr;

   public Adresse(String plz, String ort,
             String strasse, Integer hausNr) {
      this.plz = plz;
      this.ort = ort;
      this.strasse = strasse;
      this.hausNr = hausNr;
   }
   protected Adresse() {}
}
```

In den Sample-Daten …

```
// ...
Adresse a1 = new Adresse( "50678" , "Köln", "Gustav-Heinemann-Ufer", 54 );
Adresse a2 = new Adresse( "51643" , "Gummersbach", "Steinmüllerallee", 1 );
adresseRepository.save( a1 );
adresseRepository.save( a2 );
// ...
```

Ergebnis in der H2-Console



- Adresse sollte eigentlich ein Value Object zu Student sein. Als Entity ergibt das keinen Sinn. Wie kann man ein Value Object kennzeichen?

Value Object als @Embeddable

Adresse als @Embeddable

```
package thkoeln.st2.demo.entities;

import …

@Embeddable
//@Entity
public class Address {
   //@Id
   //private long id;

   private String postcode;
   //...
}
```

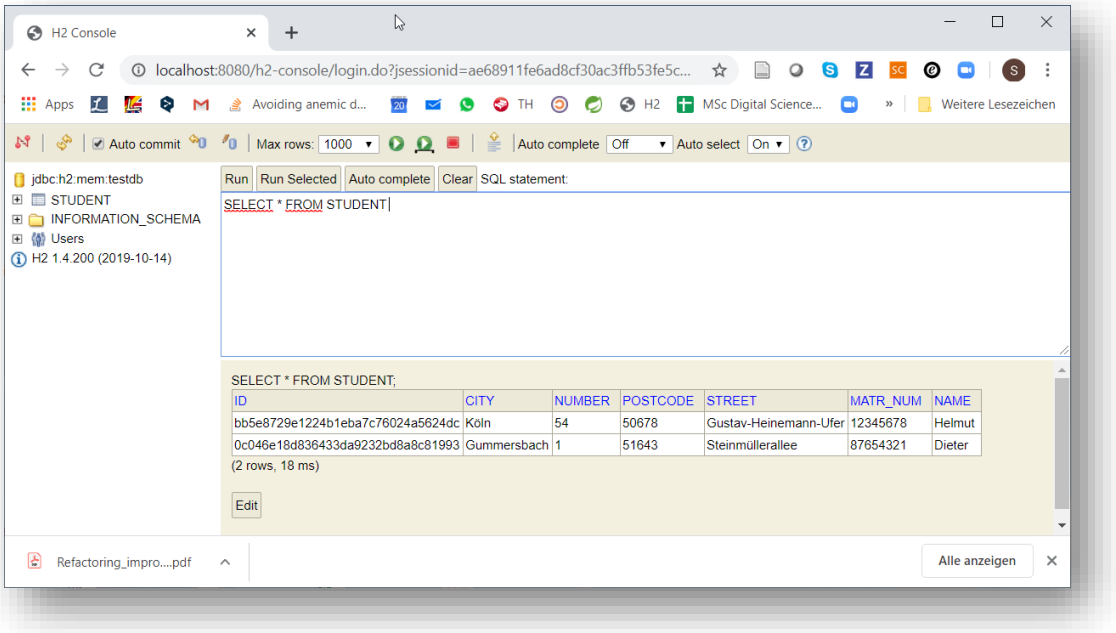
In den Sample-Daten …

```
Address a1 = new Address( "50678" , "Köln", "Gustav-Heinemann-Ufer", 54 );
Student s1 = new Student();
s1.setName( "Helmut" );
s1.setMatrNum( 12345678L );
s1.setAddress( a1 );
```

In Student …

```
@Getter
@Setter
@Embedded
private Address address;
```

Ergebnis in der H2-Console



- Value Objects kennzeichnet man als @Embeddable. Einbindung in ein Entity als @Embedded.

Value Objects in 1:n-Beziehung

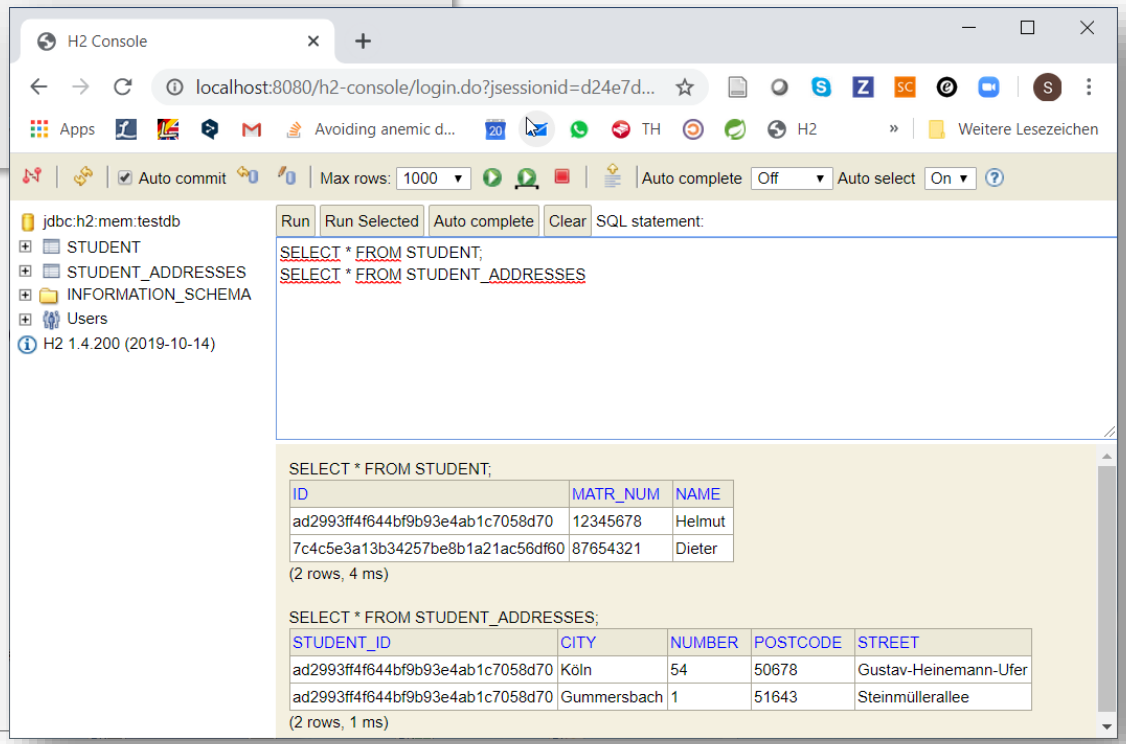
In Student …

```
//@Embedded
//private Address address;
@ElementCollection( targetClass = Address.class, fetch = FetchType.EAGER )
private Set<Address> addresses = new HashSet<Address>();
```

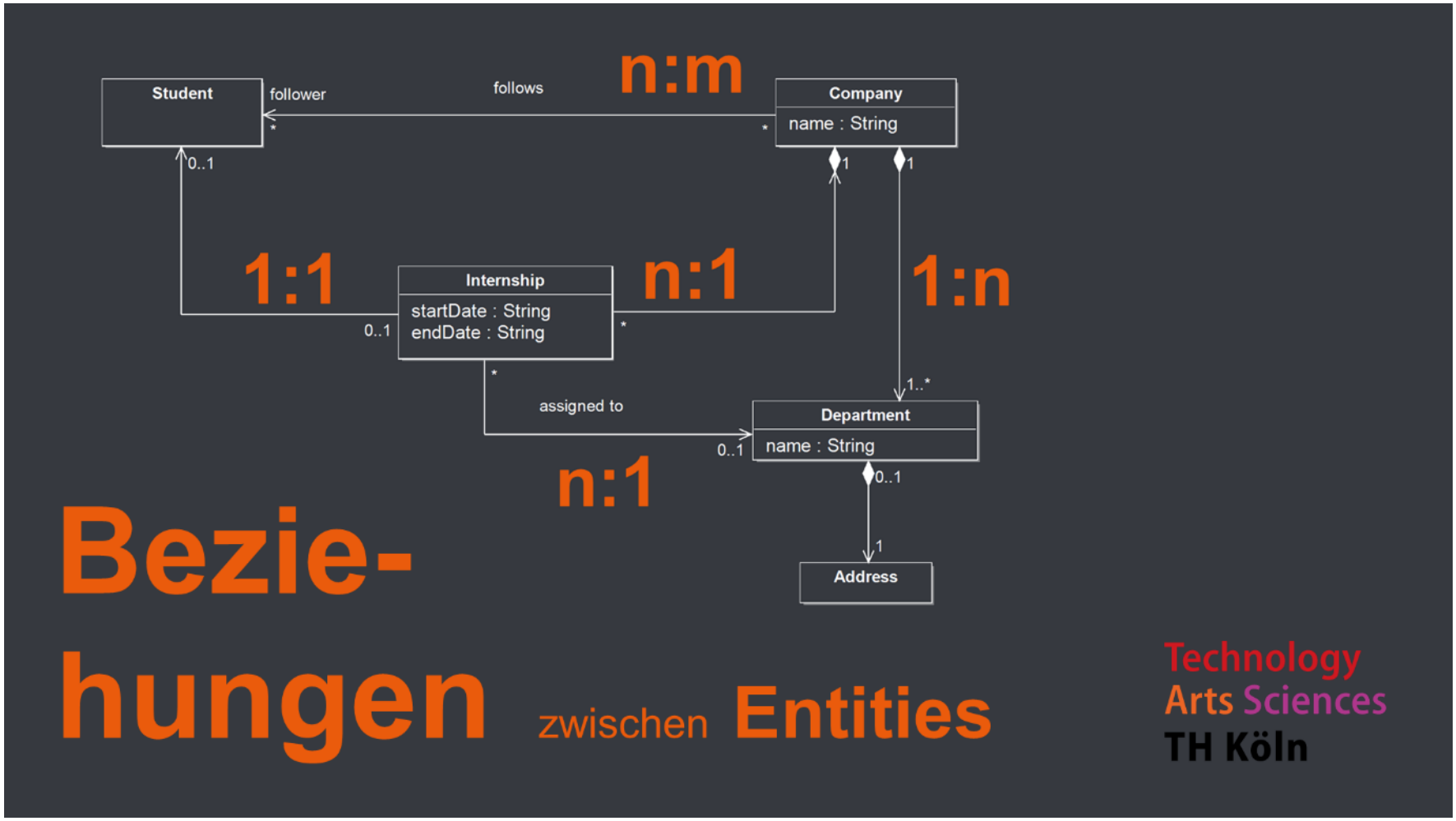
In den Sample-Daten …

```
Address a1 = new Address( "50678" , "Köln", "Gustav-Heinemann-Ufer", 54 );
Address a2 = new Address( "51643" , "Gummersbach", "Steinmüllerallee", 1 );

Student s1 = new Student();
s1.setName( "Helmut" );
s1.setMatrNum( 12345678L );
//s1.setAddress( a1 );
s1.getAddresses().add( a1 );
s1.getAddresses().add( a2 );
```

Ergebnis in der H2-Console

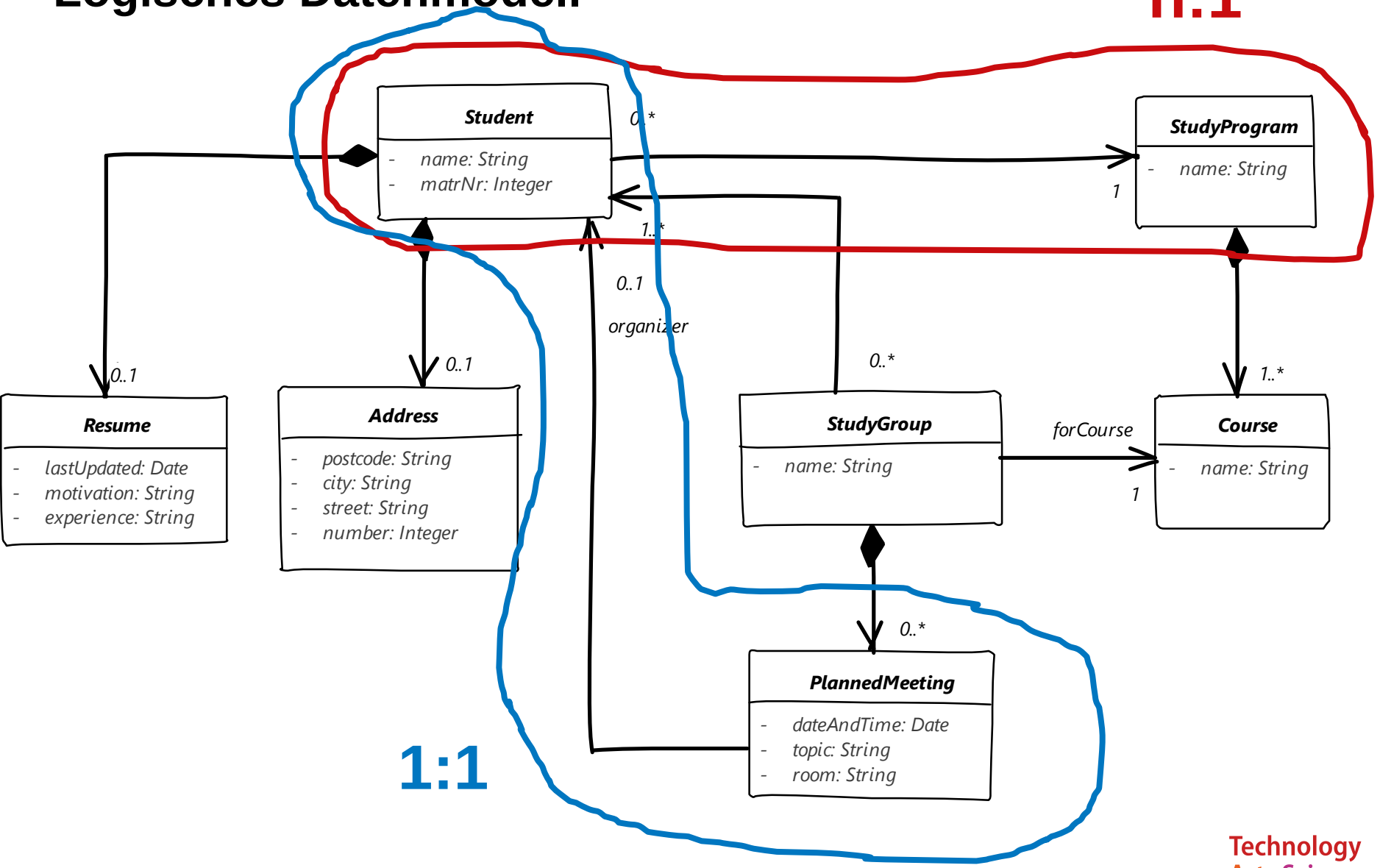


- Bei einer 1:n-Beziehung von Entity zu Value Objects nutzt man `@ElementCollection`
- Solche 1:n-Beziehungen sind allerdings in der Praxis eher selten. Typischerweise sind Value Objects als 1:1-Beziehung enthalten.



https://www.youtube.com/watch?v=GFYDgQ7QVzs

Logisches Datenmodell



n:1-Beziehung Student => StudyProgram in JPA (1)

StudyProgram

```
package thkoeln.st2.demo.entities;
import …

@Entity
@Setter
@Getter
public class StudyProgram extends AbstractEntity {
   private String name;
   public StudyProgram() {}
}
```

StudyProgramRepository

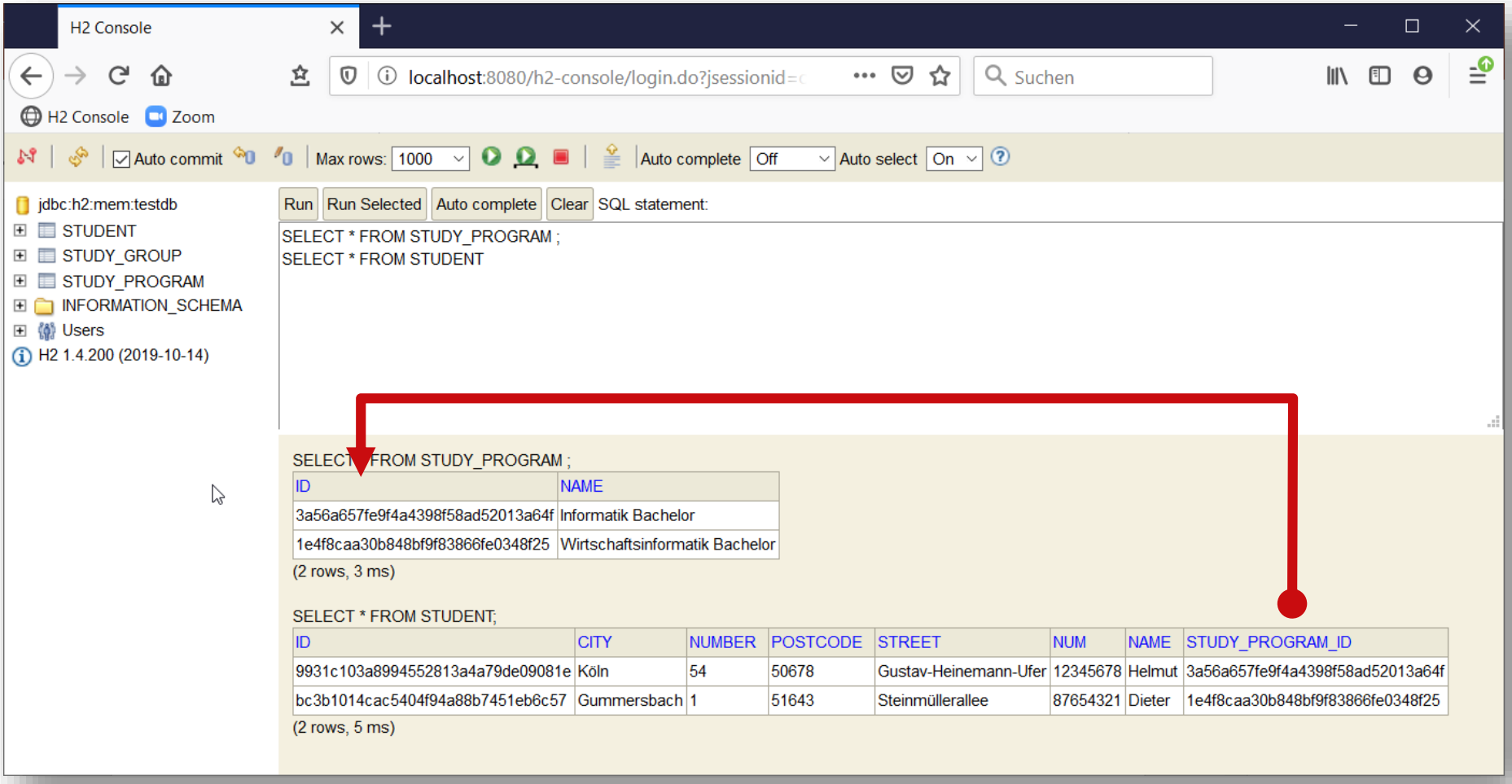
```
package thkoeln.st2.demo.repositories;
import …

public interface StudyProgramRepository extends
CrudRepository<StudyProgram, Long> {
   List<StudyProgram> findByName( String name );
}
```

DemoApplicationTests

```
@Test
public void testFindStudyProgram() {
   List<StudyProgram> list =
             studyProgramRepository.
             findByName( "Informatik Bachelor" );
   assert( list.size() == 1);
}
```

- Als Vorbereitung legen wir zunächst die beiden neuen Entities StudyProgram und StudyGroup (hier nicht gezeigt, ist aber ganz analog zu StudyProgram) mitsamt ihren Repos an. Die Entities sind erstmal „standalone“.
- In der Testklasse fügen wir je einen Test mit der jeweiligen findBy… Methode ein, um sicherzugehen, dass die Persistierung funktioniert.

n:1-Beziehung Student => StudyProgram in JPA (3)



- Die H2-Console zeigt, dass in Student dadurch ein FK auf StudyProgram angelegt wurde.

n:1-Beziehung Student => StudyProgram in JPA (2)

Student

```
@ManyToOne
private StudyProgram studyProgram;
```

CreateSampleData

```
private void createStudyPrograms() {
   sp1 = new StudyProgram();
   sp1.setName( "Informatik Bachelor" );
   studyProgramRepository.save(sp1);
}

private void createStudents() {
   Student s1 = new Student();
   s1.setName( "Helmut" );
   s1.setStudyProgram( sp1 );
   studentRepository.save( s1 );
}
```

DemoApplicationTests

```
@Test
public void testDietersStudyProgram() {
   List<Student> dieters = studentRepository.findByName( "Dieter" );
   assert( dieters.get( 0 ).getStudyProgram().getName().equals( "Wirtschaftsinformatik Bachelor" ) );
   //assert( dieter.getAddresses().size() == 0 );
}
```

- Die Beziehung wird in Student ganz einfach als `@ManyToOne` deklariert. Mit einem entsprechenden Test können wir das überprüfen.

Query der Gegenseite in n:1-Beziehung

StudentRepository

```
List<Student> findByStudyProgram( StudyProgram studyProgram );
```

DemoApplicationTests

```
@Test
public void testQueryCounterDirectionStudyProgram() {
   StudyProgram sp = studyProgramRepository.findByName( "Wirtschaftsinformatik Bachelor" ).get( 0 );
   List<Student> dieters = studentRepository.findByStudyProgram( sp );
   assert( dieters.get( 0 ).getName().equals( "Dieter" ) );
}
```

- Wir werden wesentlich häufiger den Fall haben, dass wir für einen Studenten die Studienrichtung wissen wollen. Daher ist es angemessen, dass diese Beziehungsrichtung im Code direkt abgebildet ist (siehe auch „Speziell => Allgemein“).
- Die Gegenrichtung – alle Students eines StudyProgram – braucht man weniger oft. Daher machen wir das mit einem Aufruf des Repos.

1:1-Beziehung PlannedMeeting => Student

PlannedMeeting

```java
/**
 * Planned Meeting in a StudyGroup
 */
@Getter
@NoArgsConstructor
@AllArgsConstructor
@Embeddable
public class PlannedMeeting {
   private String dateAndTime;
   private String topic;
   private String room;

   @OneToOne
   private Student organizer;

}
```

- 1:1 Beziehungen sind analog zu 1:n und n:1 Beziehungen, nur dass es kein Listentyp sein muss (wie bei 1:n)
- Tag ist `@OneToOne`

n:m-Beziehung StudyGroup => Student (1)

StudyGroup

```
public class StudyGroup extends AbstractEntity {
   private String meetsInRoom;

   @ManyToMany
   private Set<Student> students = new HashSet<Student>();

   public StudyGroup() {}
}
```

CreateSampleData

```
private void createStudyGroups() {
   StudyGroup sg1 = new StudyGroup();
   sg1.setMeetsInRoom( "0503" );
   sg1.getStudents().add( s1 );
   sg1.getStudents().add( s2 );
   studyGroupRepository.save( sg1 );

   StudyGroup sg2 = new StudyGroup();
   sg2.setMeetsInRoom( "2103" );
   sg2.getStudents().add( s2 );
   studyGroupRepository.save( sg2 );
}
```

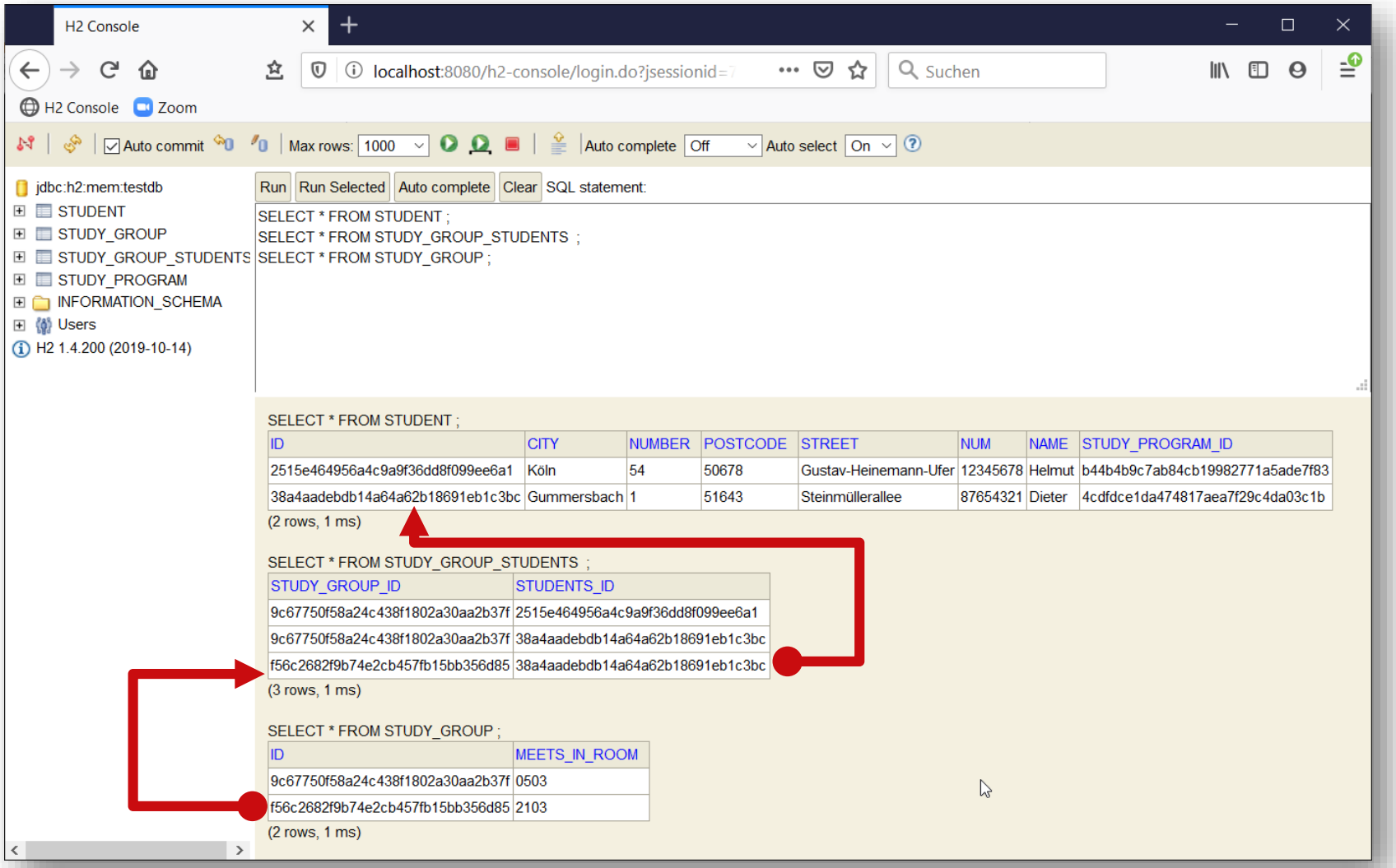
DemoApplicationTests

```
@Test
@Transactional
public void testMembersOfStudyGroup() {
   StudyGroup sg0503 = studyGroupRepository.findByMeetsInRoom( "0503" ).get( 0 );
   assert( sg0503.getStudents().size() == 2 );
   StudyGroup sg2103 = studyGroupRepository.findByMeetsInRoom( "2103" ).get( 0 );
   assert( sg2103.getStudents().size() == 1 );
}
```

- Auch n:m ist im Prinzip einfach – mittels `@ManyToMany`
- ACHTUNG, zwei häufig gemachte Fehler:

1. Der Collection-Typ (hier HashSet) muss in der Entity instanziiert werden, das macht Spring nicht für einen.
2. Man bekommt oft den Fehler „org.hibernate.LazyInitializationException: failed to lazily initialize a collection of role”. Die Antwort ist NICHT, den Fetchtype auf EAGER zu stellen, sondern die Methode in einen Transaktionskontext zu packen (daher das `@Transactional`)

n:m-Beziehung StudyGroup => Student (2)



- Die H2-Console zeigt, dass eine Zwischentabelle angelegt wurde.

Query der Gegenseite in n:m-Beziehung

StudyGroupRepository

```
List<StudyGroup> findByStudents( Student student );
```

DemoApplicationTests

```
@Test
@Transactional
public void testCounterDirectionStudyGroup() {
   Student dieter = studentRepository.findByName( "Dieter" ).get( 0 );
   List<StudyGroup> sgs = studyGroupRepository.findByStudents( dieter );
   assert( sgs.size() == 2 );
}
```

- Wir werden wesentlich häufiger den Fall haben, dass wir für einen Studenten die Studienrichtung wissen wollen. Daher ist es angemessen, dass diese Beziehungsrichtung im Code direkt abgebildet ist (siehe auch „Speziell => Allgemein“).
- Die Gegenrichtung – alle Students eines StudyProgram – braucht man weniger oft. Daher machen wir das mit einem Aufruf des Repos.

Warum

unidirektionale Beziehungen

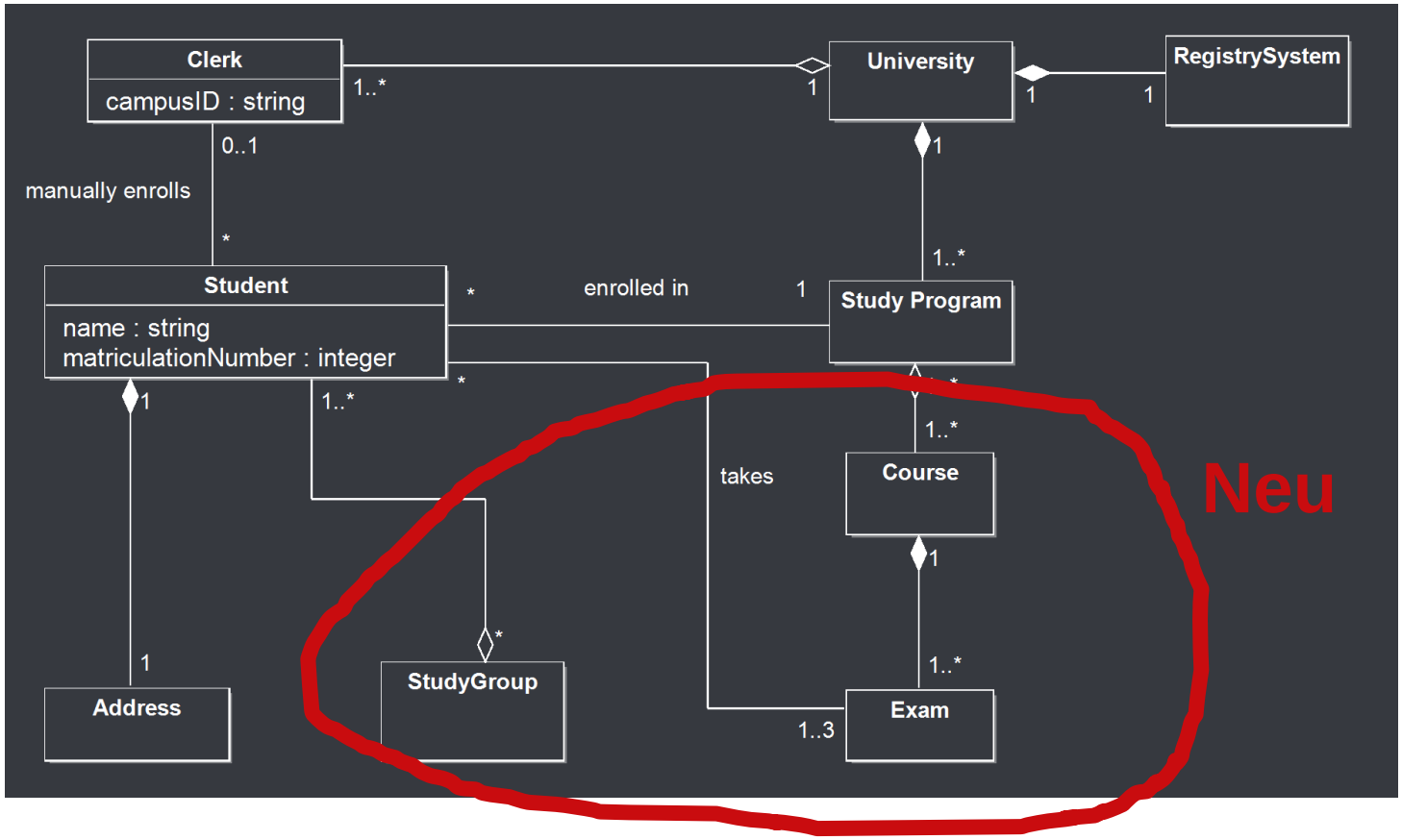
im Logischen Datenmodell?

Technology
Arts Sciences
TH Köln

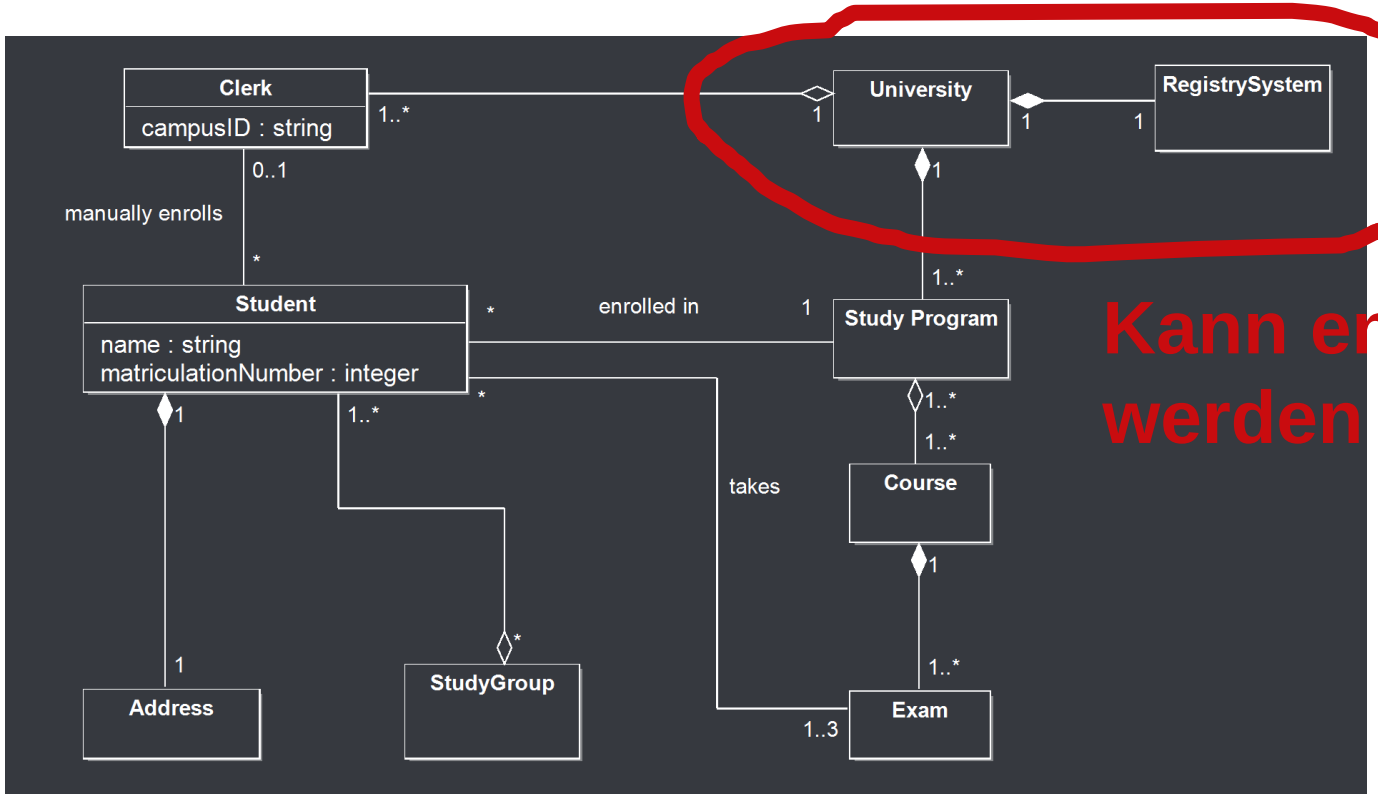
https://www.youtube.com/watch?v=RufBXPTt1SA

Unser Campus-Management-Beispiel …

- Betrachten wir das fachliche Datenmodell zu dem Campus-Management-System, das uns als Beispiel diente. Ich habe das Beispiel um einige Geschäftsobjekte angereichert, die in dem ursprünglich analysierten Text nicht enthalten waren. Diese zusätzlichen Geschäftsobjekte werden helfen, einige Aspekte besser zu erklären.



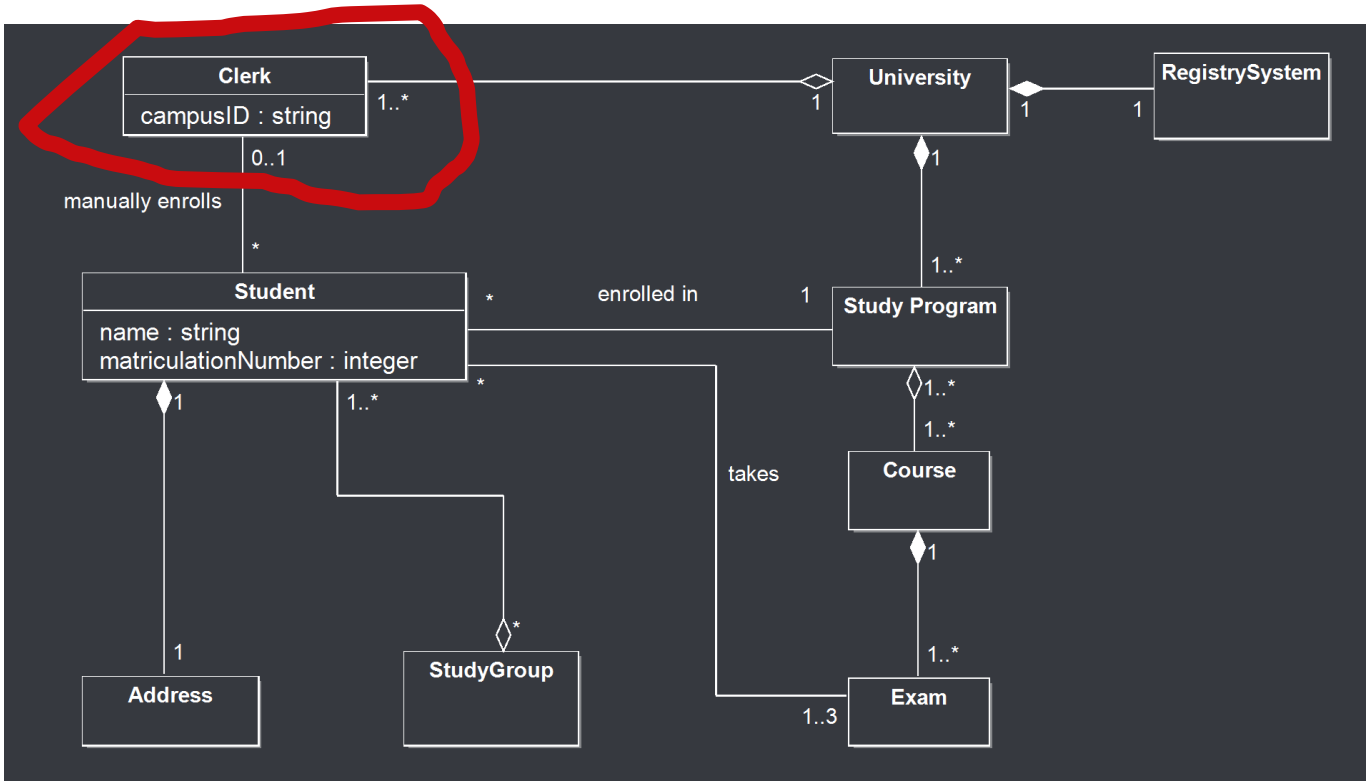
Schritt 1 der Transformation fachliches => logisches DM



- Wenn wir unser System nicht mandantenfähig auslegen wollen (d.h. es kann in vielen Universitäten eingesetzt werden, so dass “Universität” und “RegistrySystem” konfigurierbar sein müssen) – dann brauchen wir diese beiden Geschäftsobjekte nicht mehr.
- Siehe Gedankenexperiment: Hätte die entsprechende Datenbanktabelle jeweils nur eine einzige Zeile? In unserem Fall gehen wir davon aus.
- Dann können wir die beiden entfernen.

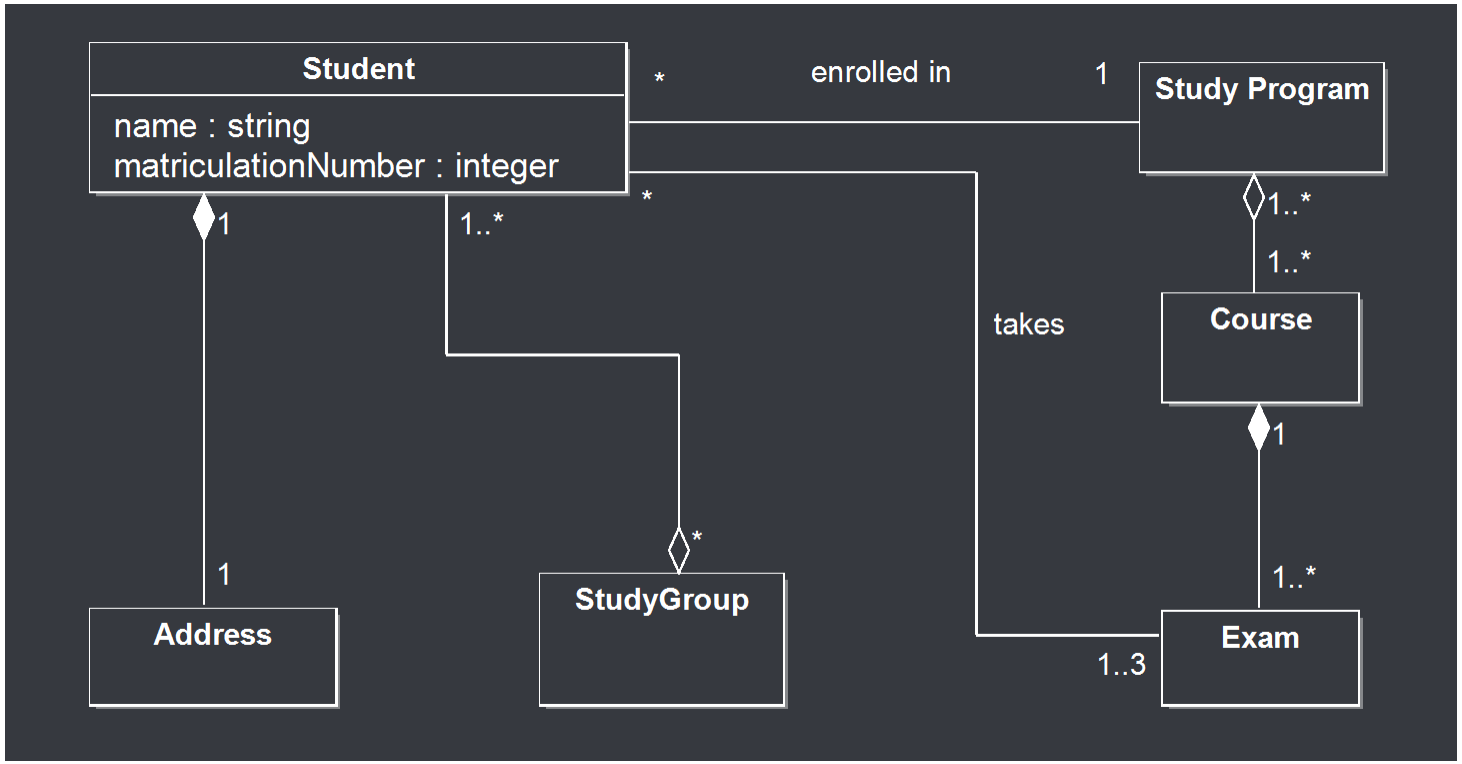
Schritt 1: Entferne überflüssige Geschäftsobjekte

Kann entfernt werden



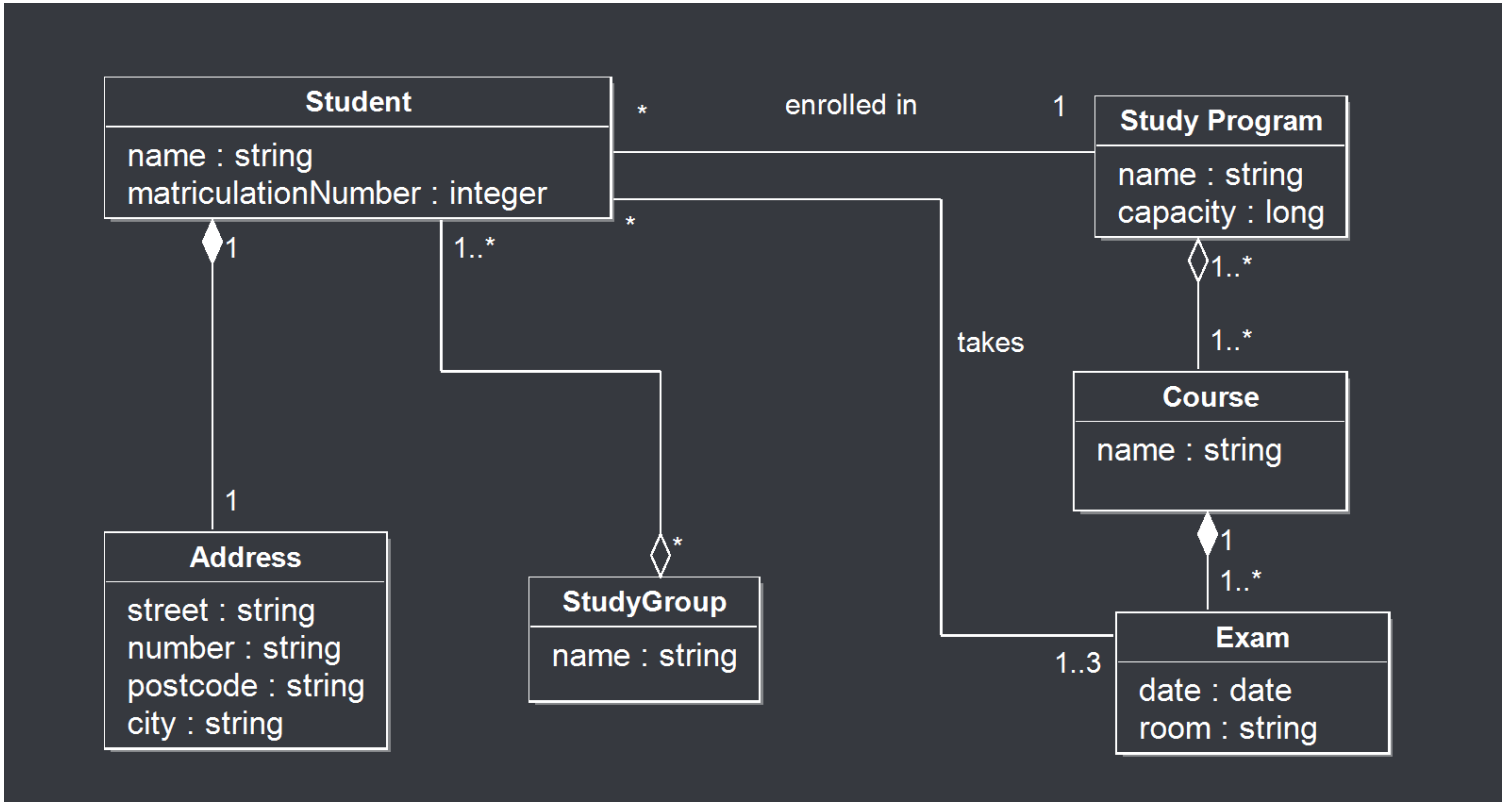
- Ebenso müssen wir nicht nachhalten, *welcher* Clerk eine Operation im System vornimmt.
- (Hätten wir ein Hoch-Sicherheits-System, z.B. eine Software in einer Strafverfolgungs-Behörde, dann müssten wir aufzeichnen, welcher Mitarbeiter welchen Zugriff unternimmt. Das ist hier aber nicht der Fall).
- Also können wir auch Clerk entfernen.

Schritt 1: Entferne überflüssige Geschäftsobjekte



- Damit ergibt sich nach Schritt 1 obiger Zustand.
- (PS: gegenüber dem früheren Video sind die Schritte 1 und 2 vertauscht. Ich entferne also erstmal die überflüssigen Klassen, dann reichere ich die Attribute an. Könnte man auch andersherum machen.)

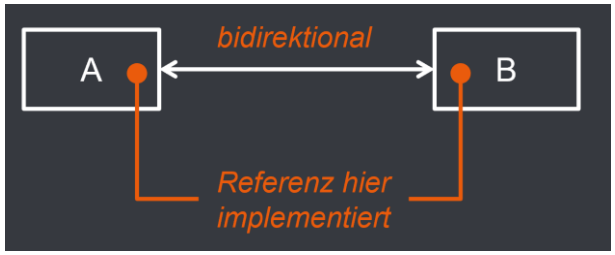
Schritt 2: Vervollständigung (insb. der Attribute)

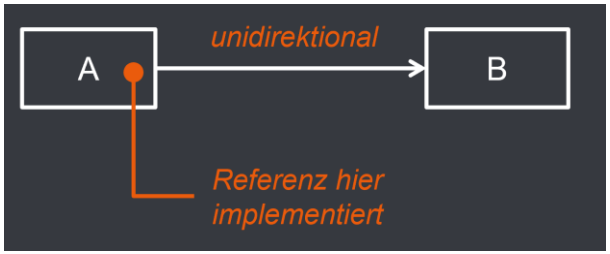


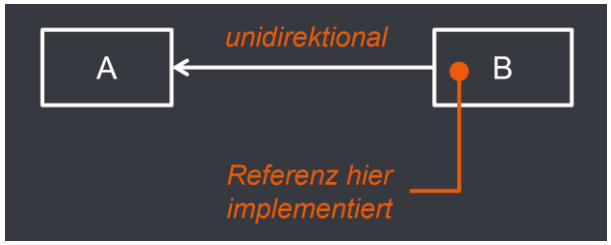
- In unserem Ursprungstext fehlen eine Menge von Informationen zu Attributen.
- Nehmen wir an, wir hätten mit der Fachseite gesprochen – dann ergäbe sich der obige Stand.
- (PS: natürlich würden wir für ein echtes produktives System noch viel mehr Informationen / Attribute brauchen. Ich halte das aber mal so einfach wie möglich, damit das Beispiel übersichtlich bleibt.)

Schritt 3: Ungerichtete vs. bidirektionale vs. unidirektionale Assoziationen

- Im **fachlichen Datenmodell** sind die Beziehungen **ungerichtet.**
- Wenn wir d.h. wir legen nicht fest, wie & wo eine Beziehung implementiert ist – wir stellen nur fest, dass beide Geschäftsobjekte mit einander in Beziehung stehen.
- Im **logischen Datenmodell** bereiten wir die Implementierung von Assoziationen vor. Jetzt müssen wir festlegen, wo eine Referenz auf die jeweils andere Seite der Beziehung implementiert wird. Dafür gibt es drei Möglichkeiten: auf beiden Seiten (**bidirektional**) bzw. jeweils nur auf einer Seite (**unidirektional**).





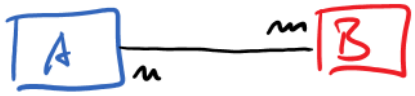


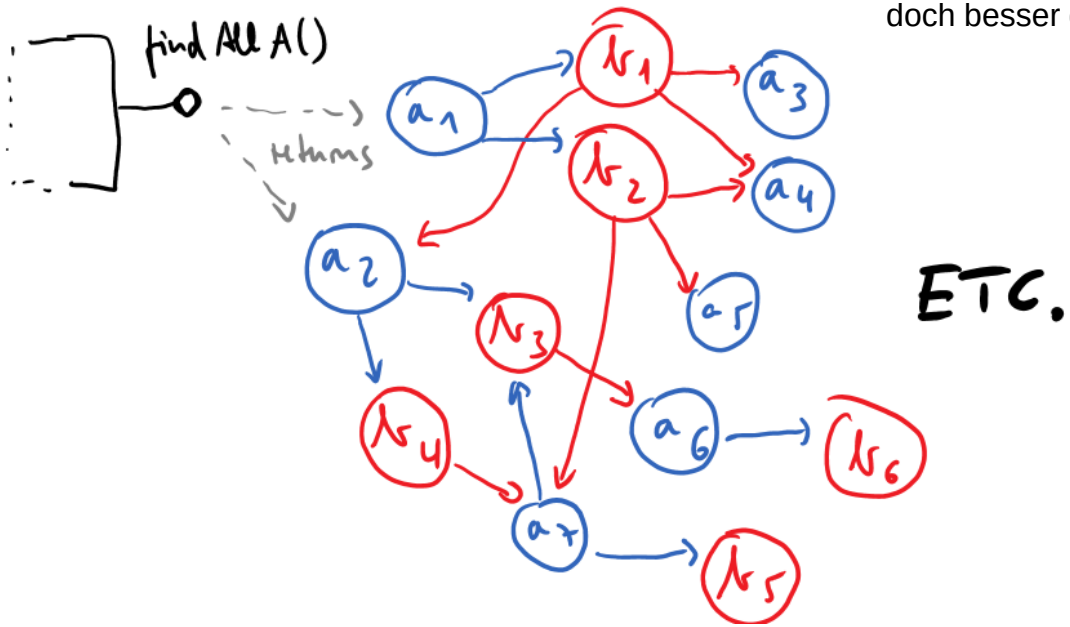
Regeln für Assoziationen

1. Stets **ungerichtete** Assoziationen implementieren
2. **Richtung**: vom Speziellen zum Allgemeinen

- “Allgemeiner” heißt dabei:
- “wird in mehr Kontexten verwendet als”
  - (Ausnahme von der Regel: Aggregate Root; das schauen wir uns in ST2 genauer an)

Was ist die Konsequenz von bidirektionalen Beziehungen?

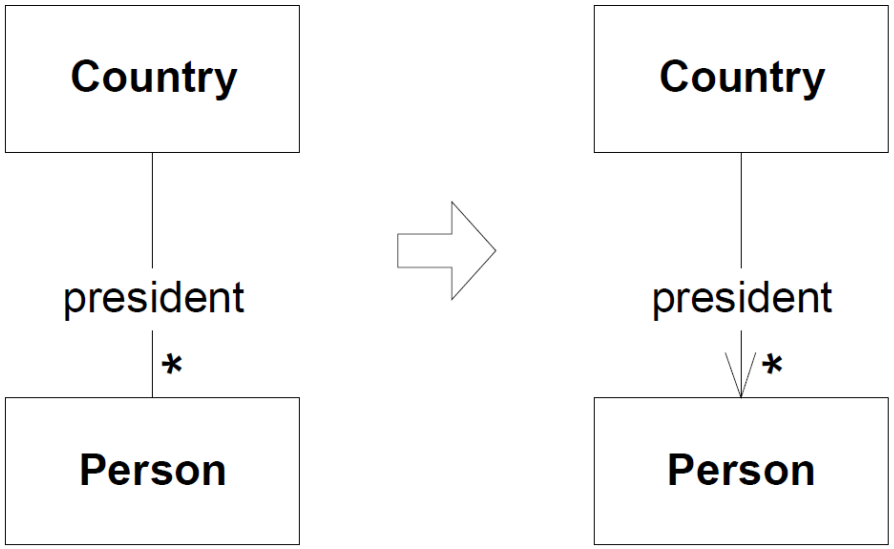




- Die Probleme bidirektionaler Beziehungen bestehen bei allen Typen (1:1, 1:n, n:1, n:m), aber bei n:m werden sie besonders augenfällig. Durch die Bidirektionalität sind die Entities eng gekoppelt – man bekommt **NIEMALS** das eine ohne das andere.
- Auf der Instanzebene kann das zu solchen regelrechten „Instanzwolken“ führen – am Ende lädt man im schlimmsten Fall **ALLE** Instanzen auf einmal, weil alle mit allen zusammenhängen.
- (Natürlich kann man diesen Effekt mit entsprechenden Annotationen verhindern, aber dann schreibt man sehr spezifische Einschränkungen in das Domainmodell hinein. Dann doch besser direkt unidirektional modellieren.

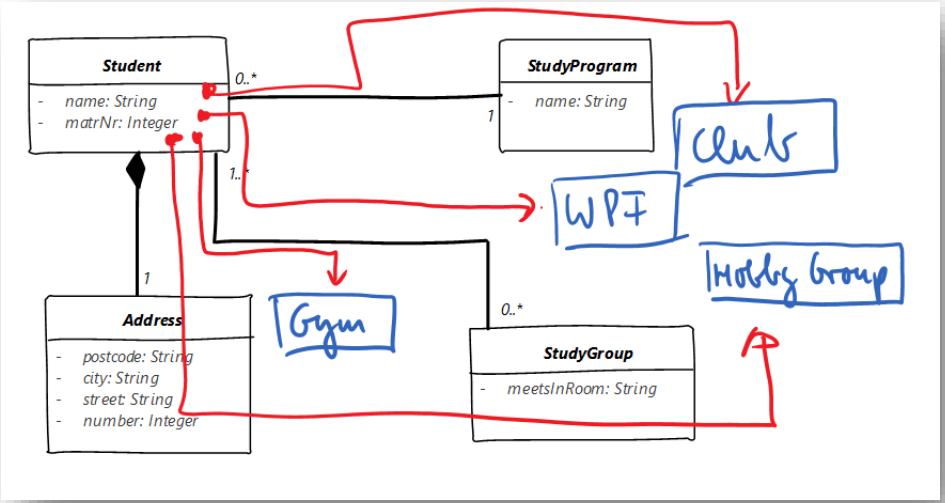
Warum vom Speziellen zum Allgemeinen (nach [Evans])

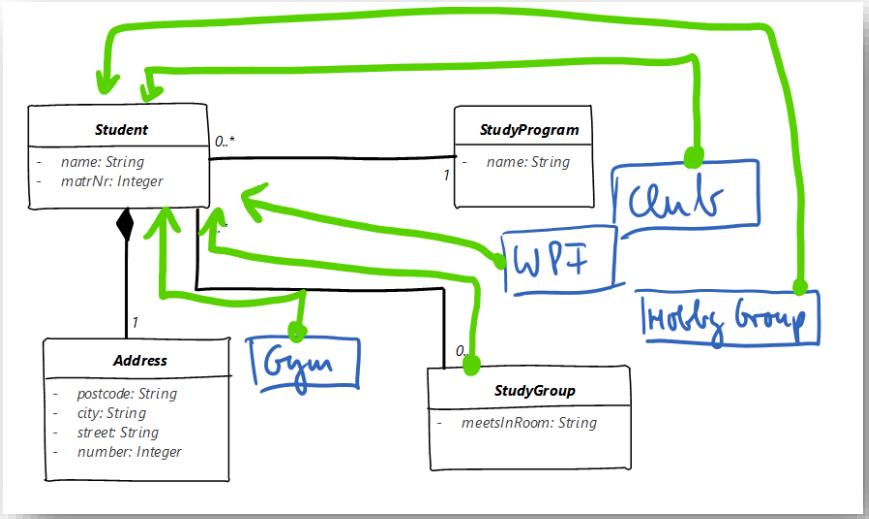
- The United States has had many presidents, as have many other countries. This is a bi-directional, one-to many relationship.
- Yet, we seldom would start out with the name “George Washington” and ask the question, “Which country was he president of?”
- Pragmatically, we can reduce it unidirectional association, traversable from **country to president**. This actually reflects insight into the domain as well as making a more practical design.
- It captures the understanding that one direction of the association is much more meaningful and important than the other.
- **It keeps the “Person” class independent of the far less fundamental concept of “President”.**



Quelle: [Evans], p. 61. Hervorhebungen und Unterteilungen in Spiegelstriche durch den Vortragenden.

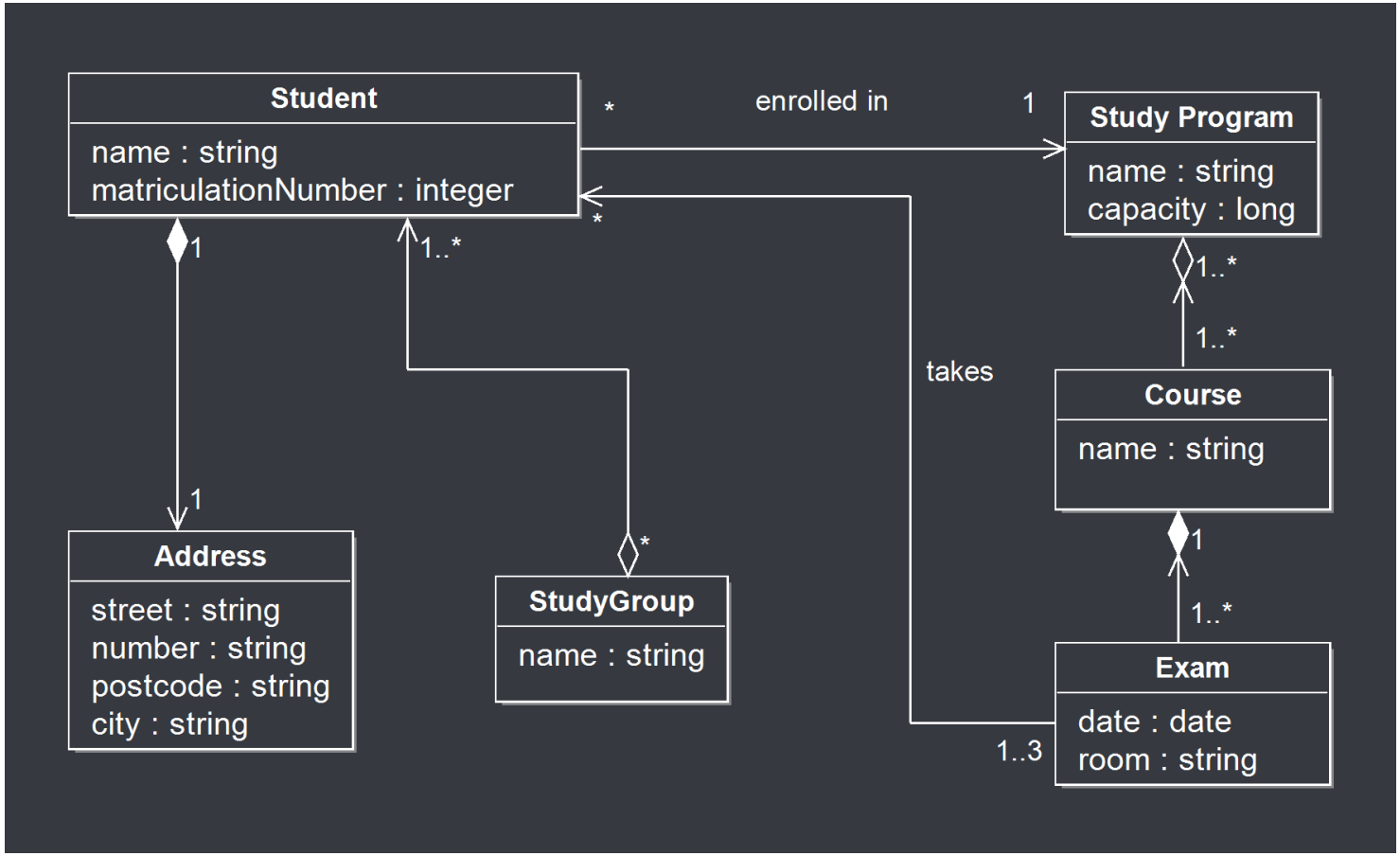
Speziell => Allgemein – warum nicht umgekehrt?





- Würde man die Beziehungen vom allgemeinen Konzept (Student) zum Speziellen (Club, WPF, HobbyGroup, Gym, …) ziehen, so entsteht eine „Gott-Klasse“, eine „Spinne im Netz“. Die Entity Student würde dann gegen „Separation of Concerns“ verstoßen: für alle Aspekte, die Club, WPF, HobbyGroup, Gym, … betreffen, muss man potentiell auch Student ändern.
- Daher ist die Richtung „Speziell => Allgemein“ sinnvoller.

Unser LDM nach Schritt 3



- Damit ergibt sich in unserem Beispiel die oben dargestellte Situation nach Schritt 3.
- Student – Address ist ein Sonderfall, da „Address“ ein Value Object (und damit eine Art „komplexes Attribut“) ist. Das diskutieren wir später noch im Rahmen der Einführung von DDD-Konzepten.