



Vienna University of Technology

Objektorientierte Modellierung

Strukturmodellierung



Business Informatics Group

*Institute of Software Technology and Interactive Systems
Vienna University of Technology*

*Favoritenstraße 9-11/188-3, 1040 Vienna, Austria
phone: +43 (1) 58801-18804 (secretary), fax: +43 (1) 58801-18896
office@big.tuwien.ac.at, www.big.tuwien.ac.at*

Willkommen zur Einheit Strukturmodellierung aus unserer Vorlesung Objektorientierte Modellierung.

Literatur

- Die Vorlesung basiert auf folgendem Buch:



UML @ Classroom:
Eine Einführung in die objektorientierte Modellierung
*Martina Seidl, Marion Brandsteidl,
Christian Huemer und Gerti Kappel*

dpunkt.verlag

Juli 2012

ISBN 3898647765

- Anwendungsfalldiagramm
- **Strukturmodellierung**
- Zustandsdiagramm
- Sequenzdiagramm
- Aktivitätsdiagramm

Inhalt

- **Klassendiagramm**
 - Klassen
 - Attribute und Operationen
 - Assoziationen
 - Schwache Aggregation
 - Starke Aggregation
 - Generalisierung
 - Zusammenfassendes Beispiel
 - Übersetzung nach Java
 - Datentypen in UML
- Objektdiagramm
- Paketdiagramm
- Abhängigkeiten

Den Schwerpunkt der Strukturmodellierung bildet das **Klassendiagramm**. Wir werden hier vor allem die Notation von Klassen in UML durchnehmen. Ich werde nicht mehr spezifisch sondern nur kurz erläutern, was Klassen und Objekte sind, weil wir das als Gegenstand der Vorlesungen objektorientierte Programmierung und Einführung in das Programmieren erachten. Wir werden die Notation von Attributen und Operationen kennenlernen, Assoziationen und Beziehungen, dann besondere Formen von Beziehungen: die schwache Aggregation, die starke Aggregation und die Generalisierung. Anschließend werden wir uns dann ein zusammenfassendes Beispiel ansehen und die Übersetzung dieses Beispiels nach Java durchnehmen. Danach werden wir noch die Datentypen besprechen.

Zusätzlich gibt es noch das **Objektdiagramm**, das ich schon während dem Klassendiagramm einstreuen werde. D.h. ich werde einmal einen kurzen Sprung und wieder zurück in den Folien machen.

Zum Abschluss der Einheit besprechen wir dann noch das **Paketdiagramm** und die **Abhängigkeiten**.

Klassendiagramm

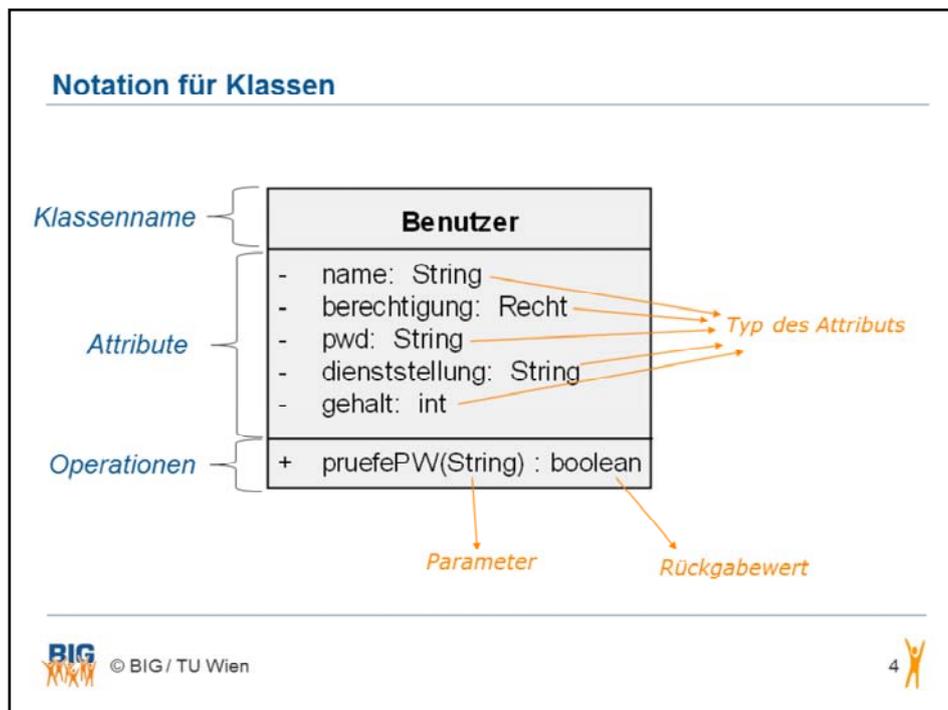
- **Klasse in UML:** Schablone, Typ
- **Objekt:** Ausprägung einer Klasse
- **Klassendiagramm**
 - Beschreibt den **strukturellen Aspekt** eines Systems auf **Typebene** in Form von Klassen, Interfaces und Beziehungen

Wir beginnen mit dem Klassendiagramm.

Was ist eine **Klasse**? In der Objektorientierung ist es die Typ-Ebene zum Beschreiben mehrerer Objekte mit der gleichen Struktur. Sehr oft ist es aber auch entscheidend, was verstehe ich unter einem Objekt? Nehmen wir als Beispiel die Klasse „Reifen“. Was sind die Instanzen von der Klasse „Reifen“? Rede ich jetzt von dem Reifenstapel, bei dem jeder einzelne Reifen eine Instanz darstellt, oder meine ich mit „Reifen“ eigentlich den Reifentyp? Dann habe ich den einzelnen physischen Reifen vielleicht gar nicht als Objekt in meiner Anwendung, was durchaus oft Sinn macht. Wenn ich ein Reifenhändler bin, werde ich kaum in der Inventur meine einzelnen Reifen erfassen können und es ist mir auch egal. Ich werde eigentlich nur den Reifentyp, beispielsweise „Semperit X10“, erfassen und von dem habe ich sieben Stück auf Lager. Das ist eines der wichtigsten Dinge, dass Sie sich immer fragen, wenn Sie eine Klasse modellieren, was sind die Objekte dieser Klasse?

Das **Klassendiagramm** beschreibt die **strukturellen Aspekte** eines Systems auf der **Typ-Ebene** in Form von Klassen, Interfaces und Beziehungen zwischen den Klassen, mit den verschiedensten Arten von Beziehungen, die wir noch kennenlernen werden.

Das **Objektdiagramm**, das man ganz selten braucht, dient eigentlich nur zur Kommunikation zwischen Implementierern oder Analysten, um beispielsweise aufzuzeigen, dass die Multiplizität doch anders ist, als man ursprünglich im Klassendiagramm angenommen hat.



Dann kommen wir zur **Notation der Klassen**.

Wenn Sie in der Übung die Notation einer Klasse vorstellen, dann wünsche ich mir eine ausführliche Beschreibung davon. Das bedeutet: eine Klasse wird als Viereck dargestellt. Das ist ja noch nicht so schwer. Doch die nächste Behauptung ist bei der Übung sehr oft falsch. Behauptet wird: Eine Klasse besteht aus drei Abschnitten. Aus wie vielen Abschnitten besteht eine Klasse? Jetzt kann man sagen, üblicherweise aus drei Abschnitten: im ersten Abschnitt befindet sich der **Name der Klasse**, und die Hinweise, ob es sich um eine abstrakte Klasse handelt, oder nicht und solche Merkmale der Klasse; im zweiten Abschnitt befinden sich die **Attribute**; im dritten Abschnitt befinden sich die **Operationen**. Die Anzahl der Abschnitten ist jedoch nach oben hin offen, d.h. es könnte noch einen vierten, einen fünften und einen sechsten Abschnitt geben. Diese Abschnitte werden selten genutzt, aber Sie machen durchaus Sinn. Man könnte beispielsweise weitere Einschränkungen angeben. D.h. ich könnte eine Einschränkung als eine zusätzliche Regel aufnehmen. Z.B. wenn die Dienststellung „Abteilungsleiter“ ist, dann muss das Gehalt größer als 5.000,- EUR sein. Das wäre eine zusätzliche Regeln, die ich im vierten Abschnitten angeben könnte. Diese Einschränkung könnte ich in natürlicher Sprache angeben, doch gibt es als Teil der UML auch die Object Constraint Language, kurz OCL, welche eine formale Definition von Einschränkungen, die nicht visualisiert werden können, erlaubt. Die OCL ist jedoch nicht Gegenstand dieser Vorlesung.

Kommen wir zurück zum Thema der Anzahl der Abschnitte. Lassen Sie sich hier auch nicht von den Tools täuschen, dass es immer der zweite Abschnitt sein muss, der die Attribute enthält, und der dritte, der die Operationen enthält. Per Default ist es zwar so, dass der zweite Abschnitt immer die Attribute enthält und der dritte immer die Operationen, doch es steht Ihnen frei, Dinge in Tools aus- und einzublenden. Wenn ich die Attribute ausblende, dann schaut es natürlich so aus, als ob die Operationen im zweiten Abschnitt wären.

Im zweiten Abschnitt befinden sich also die **Attribute** und auch hier gibt es eine Syntax, die einzuhalten ist. Als erstes folgt die Angabe des Sichtbarkeits-Operators, danach der Name des Attributs, ein Doppelpunkt : und danach steht der Typ des Attributs. Das ist entweder ein von UML vordefinierter Datentyp, ein selbst definierter Datentyp, oder eine andere Klasse. Auf die Datentypen werden wir später noch genauer eingehen.

Bei den **Operationen** im dritten Abschnitt erfolgt die Angabe wie folgt: Zuerst der Sichtbarkeits-Operator, danach der Name der Operation und in Klammern () die Inputparameter, danach ein Doppelpunkt : und dann der Rückgabewert oder Outputparameter. Bei den Inputparametern, die innerhalb der Klammern () stehen, kann ich wie in unserem Beispiel nur die Datentypen angeben. D.h. Datentyp des ersten Input-Parameters, Beistrich „“, Datentyp des zweiten Inputparameters, Beistrich „“, Datentyp der dritten Inputparameters usw. Ich kann aber für jeden Inputparameter anstatt des Datentyps auch nur den Variablennamen angeben, oder als dritte Möglichkeit ich kann beides angeben, d.h. Variablenname des ersten Parameters, Doppelpunkt :, Datentyp des ersten Parameters, Beistrich „“, Variablennamen des zweiten Parameters, Doppelpunkt :, Datentyp des zweite Parameters usw.

Attribute und Operationen

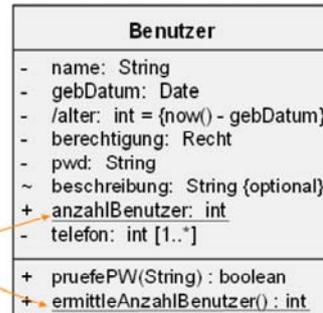
▪ Sichtbarkeiten von Attributen und Operationen:

- + ... public
- - ... private
- # ... protected
- ~ ... package (vgl. Java)

▪ Eigenschaften von Attributen:

- „/“ attributname: abgeleitetes Attribut
 - Bsp.: /alter:int
- {optional}: Nullwerte sind erlaubt
- [n..m]: Multiplizität

Klassenattribute/-operationen



Wir kommen zur Notation der **Sichtbarkeit**.

Das Plus + bedeutet „**public**“, das Minus - bedeutet „**private**“, das Nummernzeichen oder Raute # bedeutet „**protected**“ und die Tilde ~ steht für „**package**“.

Für die Sichtbarkeit gibt es auch von UML eine Definition. In der Praxis werde ich jedoch meistens die Interpretationen der Programmiersprache, die ich anwenden möchte, verwenden. Entwickle ich beispielsweise bloß in Java und ich entwickle ein plattformspezifisches Modell, dann werde ich die Tilde ~ auch so einsetzen, wie sie in Java definiert ist.

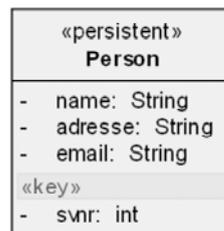
Attribute haben zusätzliche Eigenschaften, die man angeben kann. Wenn ich den Schrägstrich, Slash / vor einem Attribut verwende, dann bedeutet das, dass dieses **Attribut berechnet** werden kann. Man könnte hier die Angabe haben „/alter“ mit einem Integer-Wert. Wenn ich zusätzlich ein Attribut für das Geburtsdatum habe, dann kann ich natürlich das Alter berechnen und dann verwendet man diesen Schrägstrich /. Man könnte für ein berechnetes Attribut in geschwungenen Klammern {} auch noch die Berechnungsvorschrift angeben. In unserem Beispiel für die Berechnung des Alters könnte man die Vorschrift angeben Now minus Geburtsdatum „now () – gebDatum“, sodass man daraus das Alter berechnen kann.

Dann kommen wir zur Angabe von **optionalen Attributen**. D.h. ich gebe hinter dem Datentyp in geschwungenen Klammern {} noch an, ob bei einer Instanz ein Null-Wert erlaubt ist, oder nicht. Dies hat in der Praxis jedoch kaum Relevanz, weil ich meistens annehme, dass auch Null-Werte für Attribute erlaubt sind. Kaum jemand geht her und definiert ob Werte optional sind, oder nicht.

Man kann auch für Attribute eine **Multiplizität** im Anschluss an den Datentypen angeben. Diese Multiplizität wird in der Form von eckigen Klammern [] angegeben. Hier wird eine untere Schranke, zwei Punkte .. und die obere Schranke definiert. Habe ich z.B. ein Attribut „Telefon“ bei unserem Benutzer, könnte ich definieren, dass ich mehrere Telefonnummern angeben kann. Nämlich in unserem Fall ein Minimum von 1 und ein Maximum von Stern * oder n was für „unbeschränkt“ oder „unbounded“ steht. Es sei jedoch hier angemerkt, dass Multiplizitäten bei Attributen nicht ratsam sind, weil das meistens darauf hindeutet, dass man an Stelle eines mehrwertigen Attributs eher eine Klasse modellieren sollte. D.h. eine Klasse „Benutzer“ und eine Klasse „Telefon“ und eine Beziehung zwischen den beiden Klassen. Aber zu Beziehungen kommen wir noch später.

Ein weiterer ganz wesentlicher Punkt: Die Klasse „Benutzer“ besitzt ein unterstrichenes Attribut „anzahlBenutzer“. Wenn Sie auch in die Vorlesung Datenmodellierung gehen, dann wissen Sie, dass im Kontext von ER-Diagrammen ein unterstrichenes Attribut für ein Schlüsselattribut steht. In UML ist die Bedeutung komplett anders, da bedeutet ein unterstrichenes Attribut, dass es sich um ein **Klassen-Attribut** bzw. eine **Klassen-Operation** handelt. Ein Klassen-Attribut – vergleichbar mit „static“ in Java – ist für alle Instanzen dieser Klasse gleich. Bei unserem Beispiel mit der Klasse „Benutzer“ und dem unterstrichenen Klassen-Attribut „anzahlBenutzer“, besitzt dieses Attribut, im Falle von hundert Benutzern, bei jedem Objekt den Wert 100.

Erweiterung von UML zur Datenmodellierung



Jetzt fragen Sie sich vielleicht, warum es kein unterstrichenen **Schlüsselattribut** gibt. Das liegt daran, dass wir die Objektorientierung behandeln. In der Datenmodellierung bei den Entity Relationship Diagrammen haben wir Tupel. Die Eindeutigkeit wird festgelegt durch die Werte in diesen Tupeln. D.h. es kann keine zwei Tupel geben, die dieselben Werte haben. D.h. die Schlüsseleigenschaft wird über die Attribute definiert. In der Objektorientierung ist dem nicht so – jedes einzelne Objekt hat einen Object-Identifizier, der nicht Teil der Attribute ist. Dementsprechend kann es zwei wohldefinierte, unterschiedliche Objekte geben, die in all den Attributen genau dieselben Werte haben. Sie haben nur einen unterschiedlichen Object-Identifizier. Das ist ein wesentlicher Unterschied zwischen objektorientierter Modellierung und ER-Modellierung.

Natürlich kann man die UML auch für den Datenbankentwurf einsetzen. Doch dazu sind Erweiterungen mit sogenannten **Stereotypen** notwendig. Man könnte seinen eigenen Stereotypen „key“ als spezielle Form eines Attributs erzeugen. Stereotypen werden dann innerhalb von doppelten Spitzklammern <<>> gekennzeichnet. So gelingt es, die Schlüsselattribute als „key“ speziell auszuzeichnen. Das Problem ist, dass „key“ nicht Teil der UML selbst ist, sondern es wurde mit Hilfe des Erweiterungsmechanismus der UML definiert. „key“ ist damit auch natürlich nicht genormt. D.h. „key“ habe ich erfunden in meiner Erweiterung. Jemand anderes könnte eine andere Erweiterungsform wählen um Schlüsselattribute auszuzeichnen.

Exkurs: Identifikation von Klassen

- Linguistische Analyse der Problembeschreibung nach R.J. Abbott, Program Design by Informal English Descriptions, CACM, Vol. 26, No. 11, 1983
- **Hauptwörter** herausfiltern
- **Faustregeln**
 - Eliminierung von irrelevanten Begriffen
 - Entfernen von Namen von Ausprägungen
 - Beseitigung vager Begriffe
 - Identifikation von Attributen
 - Identifikation von Operationen
 - Eliminierung von Begriffen, die zu Beziehungen aufgelöst werden können

Auf den nächsten Folien kommen wir zu etwas, bei dem ich selbst immer hinterfrage, ob das sinnvoll ist. Aber es ist jedenfalls ein erster Ansatzpunkt für den Beginner.

Wie komme ich **aus** den Angaben, aus den **Requirements**, zu meinem **Klassendiagramm**? Und da gibt es eine Publikation „Program Design by Informal English Descriptions“ und im Wesentlichen geht es darum, Hauptwörter herauszufiltern. Man hat ein paar Faustregeln:

Eliminieren von irrelevanten Begriffen. Nur wie merke ich, was irrelevant ist? - Bauchgefühl.

Entfernen von Namen von Ausprägungen. Wenn „Christian Huemer“ vorkommt werde ich es wahrscheinlich vergessen können, „Person“ ist wichtiger.

Beseitigung vager Begriffe. Aber was ist schon vage?

Die Identifikation von Attributen sollte ich vornehmen.

Die Identifikation von Operationen aufgrund des reinen Textes stelle ich persönlich sowieso in Frage, weil ich kann in den Requirements zwar erheben, was ich mit dem Objekt machen möchte, was ich von dem Objekt erwarte, aber ob das letztendlich die Operationen sind, wage ich zu bezweifeln. Die Operationen kommen meist erst nach einem Systementwurf, wo ich mir überlege, wie denn die Klassen sich gegenseitig aufrufen.

Als letzter Punkt bei der Identifikation von Klassen sei auch noch herausgestrichen, dass ich jene Begriffe oder Hauptwörter eliminieren muss, die nicht zu Klassen werden, sondern zu Beziehungen zwischen den Klassen.

Exkurs: Identifikation von Attributen

- **Adjektive und Mittelwörter** herausfiltern
- **Faustregeln**
 - Attribute beschreiben Objekte und sollten weder klassenwertig noch mehrwertig sein
 - abgeleitete Attribute sollten als solche gekennzeichnet werden
 - kontextabhängige Attribute sollten eher Assoziationen zugeordnet werden als Klassen
- Attribute sind i.A. nur unvollständig in der Anforderungsbeschreibung definiert

Bei der **Identifikation von Attributen** steht hier: Adjektive und Mittelwörter herausfiltern. Aber ich glaube, sehr oft sind es auch Hauptwörter. Wenn ich beispielsweise sage: eine Person hat ein Alter, dann ist das kein Mittelwort, aber „Alter“ könnte durchaus ein Attribut der Klasse „Person“ sein.

Ein weiterer ganz wichtiger Punkt: Attribute beschreiben Objekte und sollten weder klassenwertig noch mehrwertig sein.

Was versteht man unter **mehrwertig**? Unter mehrwertig versteht man Attribute, bei denen die Multiplizität größer als 1 ist. Das haben wir auf der Folie Nummer 5 schon kennengelernt. Da gab es das Attribut „Telefon“ und aufgrund der Multiplizität kann ich im Attribut „Telefon“ mehrere Telefonnummern speichern. Die Forderung nach Nicht-Mehrwertigkeit muss daher übersetzt werden zu: Verzichten Sie auf die Multiplizitätsangabe. Die obere Schranke bei einem Attribut sollte immer 1 sein.

Klassenwertig wäre, wenn das Attribut selbst wieder eine Struktur besitzt, d.h. die Klasse „Person“ hat ein Attribut „Adresse“, aber das Attribut „Adresse“ besteht wieder aus „Straße“, „Hausnummer“, „PLZ“ usw. Dann wäre „Adresse“ klassenwertig. Es kann natürlich auch sein, dass ich die Adresse nur als reinen String abspeichern will, dann ist es natürlich kein Problem, denn dann hat „String“ selbst keine Struktur. Wenn ich aber die Untergliederung vornehme, dann bedeutet es klassenwertig. Darauf sollte ich verzichten bei einem Attribut. „Adresse“ wäre dann als Klasse und nicht als Attribut zu modellieren.

Abgeleitete Attribute oder berechnete Attribute sollen als solche mit Hilfe des Schrägstrich (/) gekennzeichnet werden.

Kontextabhängige Attribute sollten eher Assoziationen zugeordnet werden als Klassen. Dafür haben wir in UML die Assoziationsklasse, die wir später in der Einheit noch kennenlernen werden. Bitte verzeihen Sie mir, dass ich nicht jetzt schon auf die Assoziationsklasse vorgreife. Attribute sind im Allgemeinen nur unvollständig in den Anforderungsbeschreibungen definiert. Sehr oft müssen Sie da noch einmal nachfragen und das dann auch in der Anforderungsbeschreibung nachziehen, weil es sonst leicht zu Inkonsistenzen kommt.

Exkurs: Identifikation von Operationen

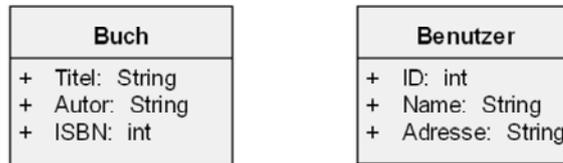
- **Verben** herausfiltern

- **Faustregeln**
 - Welche Operationen kann man mit einem Objekt ausführen?
 - Nicht nur momentane Anforderungen berücksichtigen, sondern Wiederverwendbarkeit im Auge behalten
 - Welche Ereignisse können eintreten?
 - Welche Objekte können auf diese Ereignisse reagieren?
 - Welche anderen Ereignisse werden dadurch ausgelöst?

Die **Identifikation von Operationen** sei Ihnen hier angegeben. Ich werde aber trotzdem darauf verzichten sie vorzutragen, weil das meiner Meinung nach stärker in den Systementwurf gehört.

Bsp. - Bibliotheksverwaltung

Franz Müller soll neben anderen Leuten die Bibliothek der Universität benutzen können. Im Verwaltungssystem werden die **Benutzer** erfasst, von denen eine **eindeutige ID**, **Name** und **Adresse** bekannt sind, und die **Bücher**, von denen **Titel**, **Autor** und **ISBN-Nummer** gespeichert sind.



Frage: Was ist mit Franz Müller?

Dann ein ganz einfaches Beispiel: Franz Müller soll neben anderen Leuten die Bibliothek der Universität benutzen können. Im Verwaltungssystem werden die Benutzer erfasst, von denen eine eindeutige ID, Name und Adresse bekannt sind und die Bücher, von denen Titel, Autor und ISBN-Nummer gespeichert sind.

Frage: Was ist mit Franz Müller? Franz Müller ist eine Instanz, die können wir weglassen.

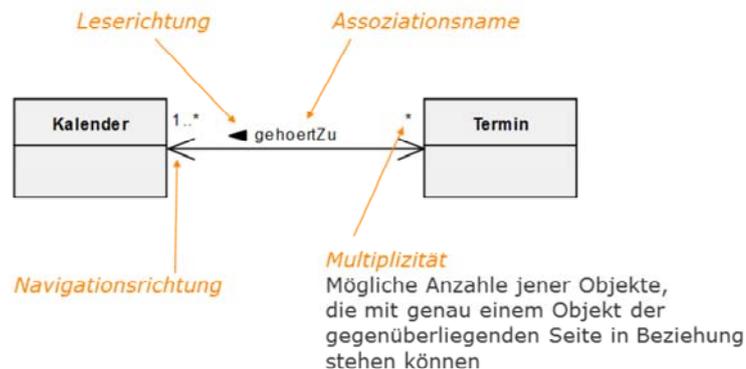
Das nächste das ich sehe ist „Leute“ und da haben wir noch einen ähnlichen Begriff „Benutzer“. „Benutzer“ ist spezifischer. Andere Leute habe ich im Moment nicht, daher nenne ich die Klasse „Benutzer“ und verzichte auf „Leute“.

„Bibliothek der Universität“: Wenn ich ein Informationssystem für eine bestimmte Einheit entwickle, dann werde ich die Bibliothek nicht als Teil des Klassendiagramms haben, weil ich davon genau ein Objekt, eine Instanz, von Bibliothek haben würde. Wenn ich jedoch ein System über den Bibliotheksverbund Österreichs entwerfen würde, dann wäre die Bibliothek natürlich auch eine eigene Klasse. In unserem Fall aber nicht.

Wir haben eben „Benutzer“ mit eindeutiger „ID“, „Name“ und „Adresse“. Die Tatsache, dass die „ID“ eindeutig ist, kann ich ohne Erweiterungsmechanismus, den ich ihn vorher vorgestellt haben, nicht notieren. Das „Buch“ hat die Attribute „Titel“, „Autor“ und „ISBN“.

Assoziation

- Assoziationen zwischen Klassen modellieren mögliche **Objektbeziehungen (Links)** zwischen den **Instanzen der Klassen**



Wir kommen nun zu den **Assoziationen**. Assoziationen stellen Beziehungen zwischen Klassen dar und es gibt eine Reihe von Notationsmöglichkeiten für Assoziationen.

In unserem Beispiel haben wir eine Beziehung zwischen der Klasse „Kalender“ und „Termin“. Die Beziehung wird durch eine Linie zwischen den beiden Klassen dargestellt.

Die Beziehung hat einen **Namen** und dieser Name wird üblicherweise in der Mitte dieser Linie darüber, darunter oder daneben dargestellt. In unserem Beispiel lautet der Name der Beziehung „gehört zu“.

Neben dem Namen der Beziehung wird mit Hilfe eines kleinen schwarzen Pfeiles die **Leserichtung** angegeben. Die Leserichtung ist modelltechnisch gesehen unbedeutend. Die Leserichtung gibt Ihnen nur an, bei welcher Klasse Sie starten müssen, wenn Sie einen korrekten Satz formulieren möchten. In unserem Beispiel zeigt der Pfeil von „Termin“ zu „Kalender“. Der richtige Satz lautet also: „Ein Termin gehört zu einem Kalender“. Umgekehrt wäre es ja falsch. „Kalender gehört zu einem Termine“ wäre ja nonsense.

Nicht zu verwechseln mit der Leserichtung ist die **Navigationsrichtung**. Die Navigationsrichtung ist modelltechnisch gesehen äußerst relevant. Die Navigationsrichtung wird an den Enden der Linie der Beziehung selbst angegeben, mit Hilfe von Pfeilspitzen. Wenn wir uns unser Beispiel ansehen, so befindet sich eine Pfeilspitze sowohl auf Seiten des Kalenders, als auch auf Seiten des Termins. Dies bedeutet, ich kann in beide Richtungen navigieren. Navigierbarkeit wird durch das Verfolgen des Pfeils ausgedrückt. Die Pfeilspitze auf der Seite von „Termin“ bedeutet, ich kann von „Kalender“ zu „Termin“ navigieren. In anderen Worten, habe ich einen Kalender, so kann ich auf die Termine in diesem Kalender zugreifen. Umgekehrt gilt auch, ich kann den Pfeil von „Termin“ zu „Kalender“ navigieren. Das bedeutet, habe ich einen bestimmten Termin, so kann ich feststellen, in welchen Kalendern dieser Termin eingetragen ist.

Zusätzlich wird auf den beiden Enden der Beziehung jeweils die **Multiplizität** angegeben. Die Multiplizität gibt die mögliche Anzahl jener Objekte an, die mit genau einem Objekt der gegenüberliegenden Seite in Beziehung stehen können. Aber zur Feststellung der Multiplizität kommen wir noch in der Folge.

Assoziation: Navigationsrichtung

- Eine gerichtete Kante gibt an, **in welche Richtung die Navigation** von einem Objekt zu seinem Partnerobjekt **erfolgen kann**
- Ein **nicht-navigierbares Assoziationsende** wird durch ein "X" am Assoziationsende angezeigt



- Navigation von einem bestimmten Termin zum entsprechenden Dokument
- Umgekehrte Richtung - welche Termine beziehen sich auf ein bestimmtes Dokument? - wird nicht unterstützt
- **Ungerichtete Kanten** bedeuten **"keine Angabe über Navigationsmöglichkeiten"**
 - In Praxis wird oft bidirektionale Navigierbarkeit angenommen
- Die Angabe von Navigationsrichtungen stellt einen Hinweis für die spätere Entwicklung dar

Sehen wir uns die **Navigationsrichtung** noch einmal genauer an.

Eine gerichtete Kante gibt an, dass ich vom Objekt zum Partnerobjekt wo die Pfeilspitze steht navigieren kann. Sollte ich von einem Objekt zum gegenüberliegenden Objekt nicht navigieren können, so muss auf der gegenüberliegenden Seite ein Kreuz angegeben werden.

Den Unterschied sieht man schön an unserem Beispiel der Beziehung zwischen „Termin“ und „HypertextDokument“. Ich kann hier vom „Termin“ zum „HypertextDokument“ navigieren, d.h. kenne ich einen Termin, so kann ich auf das für den Termin möglicherweise hinterlegte Hypertext-Dokument zugreifen. Umgekehrt: Habe ich das Hypertext-Dokument, kann ich nicht zum Termin navigieren, d.h. vom Hypertext-Dokument aus kann ich nicht feststellen, welchen Terminen dieses Hypertext-Dokument zugeordnet ist.

Laut UML-Standard muss ich die Navigierbarkeit immer mit einer Pfeilspitze oder mit einem Kreuz explizit angeben. Gebe ich beides nicht an, verwende ich praktisch eine normale Linie am Ende, so bedeutet dies, die Navigierbarkeit ist „undefined,“ es wird nicht festgelegt, ob die Beziehung in diese Richtung navigierbar ist. So sieht es der Standard vor. Dass das in der Praxis eventuell anders gehandhabt wird, sehen wir zwei Folien weiter.

Assoziation als Attribut

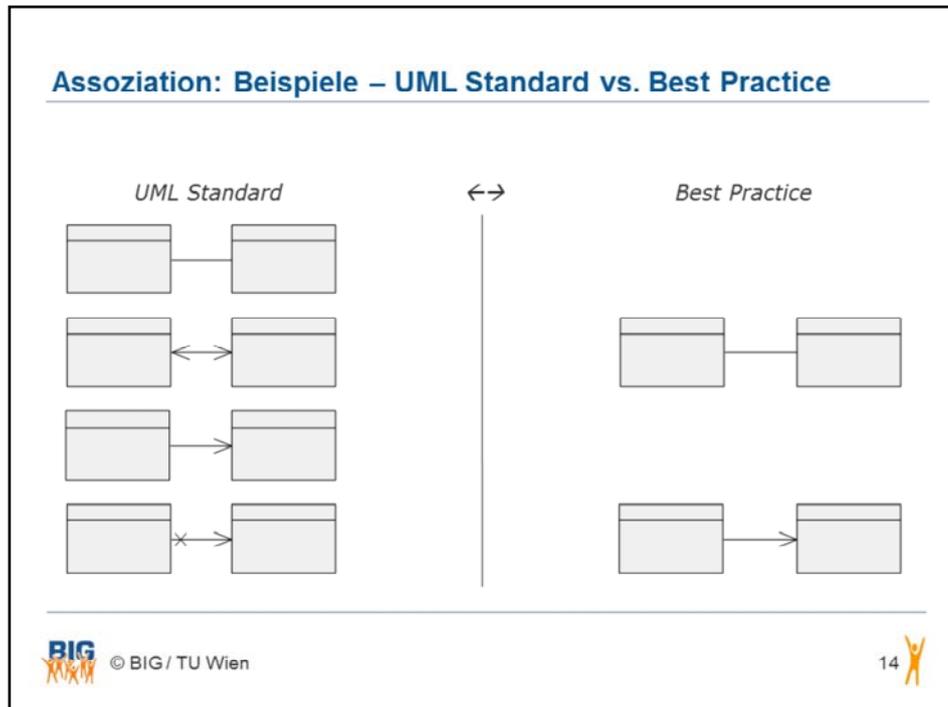
- Ein **navigierbares Assoziationsende** hat die gleiche Semantik wie ein Attribut der Klasse am gegenüberliegenden Assoziationsende
- Ein navigierbares Assoziationsende kann daher **anstatt** mit einer **gerichteten Kante** auch als **Attribut** modelliert werden
 - Die mit dem Assoziationsende verbundene Klasse muss dem **Typ** des Attributs entsprechen
 - Die **Multiplizitäten** müssen gleich sein
- Für ein navigierbares Assoziationsende sind somit **alle Eigenschaften und Notationen von Attributen** anwendbar



Anstatt eine Beziehung zwischen Klassen einzuzeichnen, könnte ich auch auf die Angabe der Beziehung verzichten und **stattdessen Attribute** in den beteiligten Klassen aufnehmen. Es sei explizit darauf hingewiesen, dass wir dies in der Lehrveranstaltung Objektorientierte Modellierung weiter nicht verfolgen werden, außer auf dieser Folie, und dass wir das auch nicht so in den Übungen verwenden werden. Trotzdem zeigen wir die äquivalente Darstellung mit Hilfe von Attributen, um die Navigierbarkeit noch einmal zu verdeutlichen.

Dementsprechend kann ich das vorherige Beispiel auch folgendermaßen darstellen. Wir haben gesagt, ich kann von „Termin“ zum „HypertextDokument“ navigieren. Dies ist äquivalent dazu, auf die Beziehung zu verzichten und an Stelle ein Attribut in der Klasse „Termin“ aufzunehmen. Dieses Attribut – in unserem Fall nennen wir es „zusatzinfo“ muss dann vom Typ der gegenüberliegenden Klasse, also vom Typ „HypertextDokument“ sein. Umgekehrt kann ich beim Beispiel auf der vorigen Folie nicht vom „HypertextDokument“ zum „Termin“ navigieren. Das Ausschließen der Navigation wurde eben durch das Kreuz am Ende der Beziehung dargestellt. In der äquivalenten Darstellung auf dieser Folie habe ich daher kein Attribut in der Klasse „HypertextDokument“. Darum kann ich von „HypertextDokument“ nicht zum „Termin“ gelangen.

Assoziation: Beispiele – UML Standard vs. Best Practice



Ungerichtete Kanten bedeuten im UML Standard „keine Angabe über die Navigationsmöglichkeit“, „undefined“, „nicht definiert“. D.h., wenn Sie ein Klassendiagramm entsprechend UML Standard zeichnen sollen und Sie sollen die Navigationsmöglichkeiten angeben, was wir in dieser Lehrveranstaltung unterstellen, dann müssen Sie immer am Ende entweder einen Pfeil oder ein Kreuz haben. Das Problem, das Sie vielleicht schon einmal erkannt haben, wenn Sie schon einmal ein Klassendiagramm gesehen haben, ist, dass ganz selten ein Pfeil oder ein Kreuz im Klassendiagramm eingezeichnet wird. In der Praxis wird nämlich folgendes angenommen: Wenn eine ungerichtete Kante existiert, also einfach eine Linie zwischen zwei Klassen, dann unterstellen wir Navigation in beide Richtungen. Und nur dann, wenn ich Navigation in nur eine Richtung haben möchte, dann verwende ich einen Pfeil, verzichte aber auf das Kreuz am anderen Ende der Seite.

Entsprechend dem UML Standard heißt das erste Beispiel auf der linken Seite: Navigierbarkeit in beide Richtungen „undefined“.

Das zweite Beispiel auf der linken Seite bedeutet: In beide Richtungen navigierbar. Im Best Practice wird die beidseitige Navigierbarkeit aber meistens so dargestellt, wie das erste Beispiel auf der rechten Seite. Es wird im Best Practice unterstellt, dass ich in beide Richtungen navigieren kann, sobald ich eine einfache Linie zwischen den beiden Klasse habe.

Im UML Standard würde das dritte Beispiel links bedeuten: Navigierbarkeit von der linken zur rechten Klasse, aber in die umgekehrte Richtung „undefined“.

Das letzte Beispiel links würde bedeuten: Von links nach rechts navigierbar, aber umgekehrte ist die Navigation ausgeschlossen. Im Best Practice wird der letzte Fall zwar durch einen Pfeil angegeben, jedoch verzichtet man auf die Angabe des Kreuzes auf der anderen Seite.

Was Sie jetzt genau sehen ist, dass das komplett andere Interpretationen ist. Diese beiden Formen auf der rechten Seite gibt es im Standard auch, die Bedeutung wie sie in der Praxis angenommen wird, ist aber anders als sie im Standard zu verwenden ist. Einerseits könnte man sagen, wir unterrichten nur den Standard. Das hilft Ihnen aber nicht weiter, wenn man in der Praxis ein Diagramm sieht, wo es anders modelliert ist. Daher sollte man auch mit dem umgehen können. Es ist mir wichtig, beides festzuhalten. Bei den meisten unserer Beispiele wird es Ihnen offen gelassen, wenn Sie ein Beispiel selbst modellieren müssen, ob Sie es Best Practice machen oder nach UML Standard. Sehr viele unserer Angaben, wo bereits ein Beispiel angegeben ist, sind nach Best Practice modelliert. Wenn es bei der Prüfung oder für das Übungsblatt explizit verlangt wird, dann wird es auch entsprechend angegeben. Dann müsste dort stehen, es wurde nach Best Practice gezeichnet oder nach UML Standard. Also entweder es wird Ihnen in der Lösung offen gelassen nach Best Practice oder UML Standard zu modellieren, oder wir geben es immer genau, explizit an.

Assoziation: Multiplizität

- **Bereich:** "min .. max"
- **Beliebige Anzahl:** "*" (= 0.. *)
- **Aufzählung möglicher Kardinalitäten** (durch Kommas getrennt)
- **Defaultwert:** 1

<i>genau 1:</i>	1
<i>>= 0:</i>	* oder 0..*
<i>0 oder 1:</i>	0..1 oder 0,1
<i>fixe Anzahl (z.B. 3):</i>	3
<i>Bereich (z.B. >= 3):</i>	3..*
<i>Bereich (z.B. 3 - 6):</i>	3..6
<i>Aufzählung:</i>	3,6,7,8,9 oder 3, 6..9

Wir kommen zur Angabe der **Multiplizität**.

Wenn ich am Assoziationsende eine Multiplizität angebe, oder auch bei einem Attribut, wie wir es vorher gesehen haben, in Form von eckigen Klammern [], dann gelten folgende Regelungen.

Die Angabe von nur 1, bedeutet „genau 1“.

Der Stern * steht für „unbounded“ oder „unbeschränkt“. Zum Stern * äquivalent ist das n. Also Sie können sich aussuchen, ob Sie Stern * oder n schreiben. Geben Sie nur einen Stern * an, so bleibt die untere Grenze undefiniert und es muss 0 angenommen werden. Daher bedeutet Stern *, 0 bis mehrere. Alternativ dazu könnte man auch die Untergrenze immer explizit angeben, das wäre dann 0 als Untergrenze/Minimum, dann der Bereichsoperator (zwei nacheinander folgende Punkte ..), dann das Maximum, in diesem Falle Stern * oder „unbeschränkt“. D.h. beide Darstellungen, nur Stern *, oder 0 bis Stern * (0..*) sind äquivalent.

Wenn ich angeben will 0 oder 1, dann kann ich das definieren mit Minimum 0, Bereichsoperator (zwei aufeinanderfolgende Punkte ..) und dann 1 (0..1).

Oder ich kann alle möglichen Werte, getrennt durch einen Beistrich, aufzählen und dementsprechend könnte ich auch „0,1“ schreiben.

Die Angabe von nur 3, bedeutet „genau 3“.

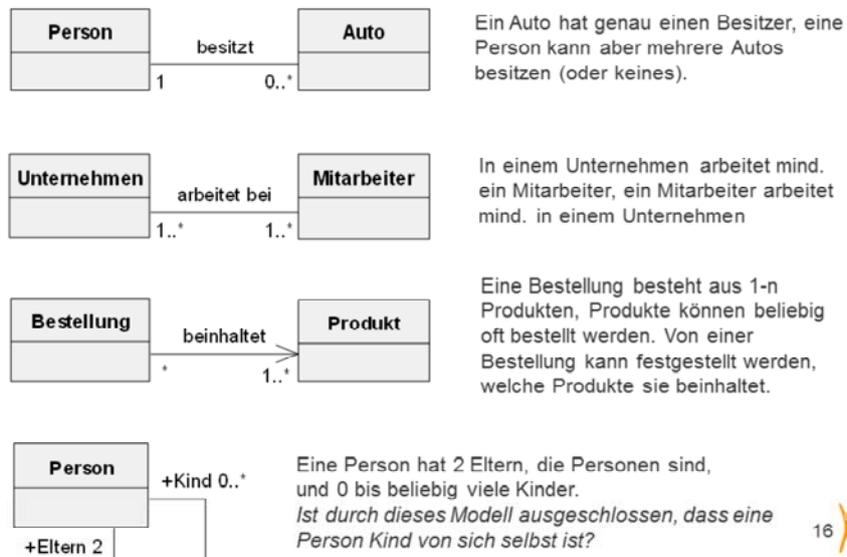
Bei einem Bereich größer als 3 hab ich wieder Minimum 3, Bereichsoperator .. und Maximum „unbeschränkt“ in Form des Stern *.

Für den Bereich 3 bis 6 gebe ich an, Minimum 3 , Bereichsoperator .. und das Maximum 6.

Mit einer Aufzählung kann ich alle möglichen Werte angeben, beispielsweise 3, 6, 7, 8, 9.

Letzeres könnte ich auch aber alternativ darstellen in dem ich nämlich zwei Alternativen biete und die zweite Alternative durch einen Bereich angebe. Also 3, oder, getrennt durch einen Beistrich, der Bereich von Minimum 6 bis Maximum 9.

Assoziation: Beispiele



Soweit die Syntax der Multiplizitäten. Aber jetzt wird es entscheidend. Wie beschrifte ich die Multiplizitäten? Wie bilde ich richtige Sätze zum **Beschriften der Multiplizitäten**?

Hier im ersten Beispiel besteht eine Beziehung zwischen „Person“ und „Auto“. Man könnte sagen, mehrere Personen können auch mehrere Autos besitzen. Ein Satz der nicht wirklich falsch ist, der mich nur nicht weiterbringt bei der Beschriftung der Multiplizitäten. Und jetzt bitte verinnerlichen: Ich muss zur Beschriftung einer binären Assoziation, d.h. an einer Assoziation, an der zwei Klassen beteiligt sind, genau so viele Sätze zum Beschriften bilden, nämlich genau zwei Sätze und jeder Satz - als Faustregel - beginnt mit dem Wort „ein“ oder „eine“. Wenn ich das erste Beispiel, bei dem jetzt links ein 1er steht, beschriften möchte, beginne ich immer auf der gegenüberliegenden Seite. Ich sage „Ein Auto wird von wie vielen Personen besessen?“ – In unserem Beispiel von genau einer. „Ein Auto wird von einer Person ...“ – daher mit 1 beschriften – „... besessen.“ Umgekehrt, wenn ich die rechte Seite beschriften möchte, sage ich „Eine Person besitzt wie viele Autos?“ – „0 bis mehrere Autos.“

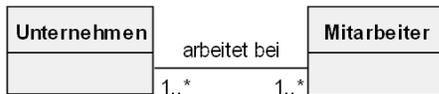
Schauen wir uns das zweite Beispiel an. Wenn ich hier die linke Seite beschriften möchte sage ich: „Ein Mitarbeiter arbeitet in wie vielen Unternehmen?“ – „In mindestens einem ...“, weil sonst wäre er kein Mitarbeiter, „... aber eventuell auch in mehreren“. Für die Beschriftung der rechte Seite gilt: „In einem Unternehmen arbeitet mindestens ein Mitarbeiter, aber auch eventuell mehrere Mitarbeiter.“

Zum dritten Beispiel: „Eine Bestellung beinhaltet wie viele Produkte?“ – „Ein Produkt oder mehrere Produkte.“ In diesem Fall kann ich nur von der Bestellung zum Produkt navigieren, d.h. ich kann feststellen, welche Produkte bei einer Bestellung enthalten sind, ich kann aber nicht vom Produkt zur Bestellung navigieren. Wenn ich vom Produkt nicht zu den Bestellungen navigieren kann, in denen dieses Produkt enthalten ist, so ist auch die Multiplizität auf der linken Seite nicht wirklich entscheidend. Der Vollständigkeit halber sei sie jedoch hier im Beispiel schon angegeben: „Ein Produkt kann in keiner oder in mehreren Bestellungen vorkommen“. Ich werde hier die Multiplizität nur dann aufnehmen, wenn ich vielleicht weiß, das später noch einmal Änderungen in der Navigationsrichtung erfolgen könnten.

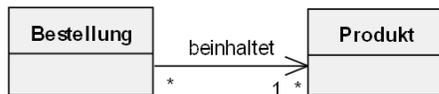
Assoziation: Beispiele



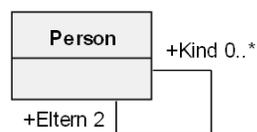
Ein Auto hat genau einen Besitzer, eine Person kann aber mehrere Autos besitzen (oder keines).



In einem Unternehmen arbeitet mind. ein Mitarbeiter, ein Mitarbeiter arbeitet mind. in einem Unternehmen



Eine Bestellung besteht aus 1-n Produkten, Produkte können beliebig oft bestellt werden. Von einer Bestellung kann festgestellt werden, welche Produkte sie beinhaltet.



Eine Person hat 2 Eltern, die Personen sind, und 0 bis beliebig viele Kinder.
Ist durch dieses Modell ausgeschlossen, dass eine Person Kind von sich selbst ist?

16 

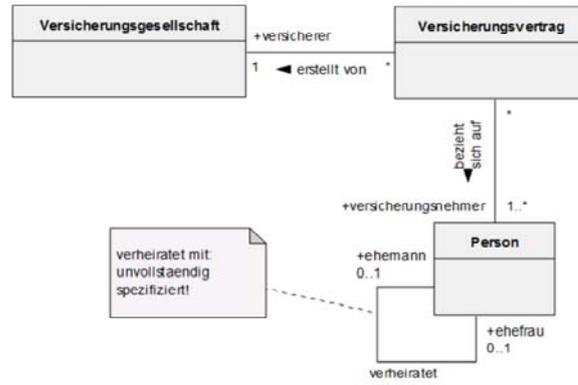
Wir sprechen, wie in der Datenmodellierung, sehr oft auch von sogenannten **1:1-Beziehungen**, **1:n-Beziehungen**, und **m:n-Beziehungen**. Das ist auch durchaus üblich bei Klassendiagrammen. Jetzt ist das aber gar nicht so leicht zu sehen, wenn die Multiplizität in Form von Bereichsangaben gemacht wird. Aber man geht immer vom Maximum bei Bereichsangaben aus. D.h., das oberste Diagramm wäre ein Beispiel für eine 1:n-Beziehung. Das zweite Diagramm wäre ein Beispiel für eine n:m-Beziehung. Im dritten Diagramm gibt es zwar auch eine n:m-Beziehung, diese ist aber nur in eine Richtung navigierbar.

Bei den ersten drei Beispielen handelt es sich um **binäre Assoziationen**, weil genau zwei Klassen beteiligt sind. Das ist die mit Abstand am häufigsten vorkommende Art von Beziehung. Es gibt auch die **unären Beziehungen**. D.h. eine Klasse steht mit sich selbst in Beziehung.

Im letzten Beispiel möchte ich die unäre Eltern-Kind-Beziehung festhalten. In diesem Fall, aber auch sonst, macht es insbesondere Sinn, die **Rollen an den Beziehungsenden** festzuhalten und zu beschriften. Wenn ich jetzt sage „Eine Person hat genau zwei Eltern“, dann beschrifte ich hier am unteren Ende der Beziehung nicht nur die Beziehung mit der entsprechenden Multiplizität, sondern auch mit der Rolle, die eingenommen wird „Eltern“. Und umgekehrt, auf der anderen Seite der Beziehung gilt: Eine Person hat 0 bis mehrere Kinder und ich gebe auch hier wieder die Rolle an. Die Frage lautet nun, wieso steht ein „+“ vor Kind und ein „+“ vor Eltern – was bedeutet denn das? Das ist der **Sichtbarkeitsoperator** „public.“ Die Sichtbarkeitsoperatoren kennen Sie ja von den Attributen, aber da ja jede Beziehung auch durch ein Attribut ausgedrückt werden könnte, und wie wir später sehen, bei der Code-Generierung auch in einem Attribut resultiert, dann stellt sich natürlich die Frage der Sichtbarkeit von diesem Attribut. Und dementsprechend ist es sinnvoll, den Sichtbarkeitsoperator am Ende der Beziehung anzugeben. Die Angabe eines Sichtbarkeitsoperators ist unabhängig davon, ob es sich um unäre Beziehungen oder eine binäre Beziehungen handelt. D.h. es würde auch bei den obigen Beispielen dieser Folie Sinn machen, ein Plus + oder ein Minus – am Ende der Assoziation hinzuschreiben.

Assoziation: Rollen

- Es können die **Rollen** festgelegt werden, die von den einzelnen Objekten in den Objektbeziehungen gespielt werden



Wir haben hier noch ein Beispiel zu **Rollen**. Rollen kann ich durchaus auch in binären Beziehungen verwenden.

Hier habe ich einen „Versicherungsvertrag“ und eine „Versicherungsgesellschaft“. Eine Versicherungsgesellschaft erstellt 0 bis mehrere Versicherungsverträge. Ein Versicherungsvertrag ist von einer Versicherungsgesellschaft erstellt und in dieser Beziehung „erstellt von“ nimmt die Versicherungsgesellschaft die Rolle „Versicherer“ ein.

Beim Überleiten nach Java werden Sie dann die Bedeutung sehen und dass es vorteilhaft ist, immer Rollen anzugeben. Bei der Übersetzung entsteht ja im Versicherungsvertrag dann ein Attribut, das eine Referenz auf die Versicherungsgesellschaft haben wird. Und jetzt ist die Frage, wie nenne ich das Attribut? Und dieses Attribut wird danach den Namen der Rolle „Versicherer“ übernehmen und es wird auch „public“ sein, weil ich hier ein „+“ am Ende der Beziehung als Sichtbarkeitsoperator angegeben habe. D.h. die Angabe von Rollen macht durchaus Sinn, weil es nachher auch bei der Übersetzung zu Code verwendet wird.

Ein Versicherungsvertrag bezieht sich auf mindestens eine, aber eventuell auch mehrere Personen, die haben die Rolle des „Versicherungsnehmers“. Umgekehrt, eine Person schließt 0 bis mehrere Versicherungsverträge ab.

Dann habe ich in diesem Beispiel noch eine unäre Beziehung von und zur Klasse „Person“. Eine Person – als Mann – hat entweder 0 oder 1e Ehefrau. Umgekehrt, eine Person – als Frau – hat entweder 0 oder 1en Ehemann. Person ist also eine Klasse und diese Klasse hat eine Beziehung mit sich selbst. Bedeutet das nun auch, dass die Objekte von Person eine Beziehung mit sich selbst haben müssen? Sie wissen sicher, dass man sich nicht selbst heiraten kann und daher ist die logische Antwort „nein“. Um den Zusammenhang zwischen Klassen und Objekten besser zu verstehen, springen wir kurz zum Objektdiagramm.

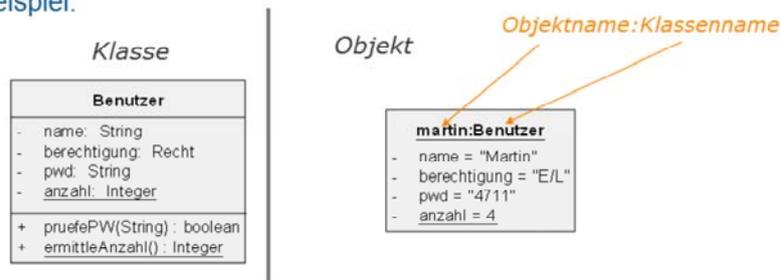
Inhalt

- Klassendiagramm
 - Klassen
 - Attribute und Operationen
 - Assoziationen
 - Schwache Aggregation
 - Starke Aggregation
 - Generalisierung
 - Zusammenfassendes Beispiel
 - Übersetzung nach Java
 - Datentypen in UML
- **Objektdiagramm**
- Paketdiagramm
- Abhängigkeiten

Sehen wir uns also kurz das **Objektdiagramm** an.

Objektdiagramm

- Beschreibt den strukturellen Aspekt eines Systems auf Instanzebene in Form von Objekten und Links
- Momentaufnahme (snapshot) des Systems – konkretes Szenario
- Ausprägung zu einem Klassendiagramm
- Eigentlich eine »Instanzspezifikation«
- Prinzipiell kann jede Diagrammart auf Instanzebene modelliert werden
- Beispiel:



72

Das Objektdiagramm beschreibt die strukturellen Aspekte eines Systems auf **Instanzebene**. Es enthält also keine Klassen, sondern **Objekte** und **Links** zwischen Objekten. Es ist eine Momentaufnahme des Systems.

Auf dieser Folie stellen wir die Notation einer Klasse mit der Notation eines Objekts gegenüber. Wir haben also links die Klasse „Benutzer“ und rechts eine Instanz dieser Klasse, das Objekt „martin“, der ein Benutzer ist. Die Angabe einer Klasse und eines Objektes sind sehr ähnlich. Die Darstellung eines Objekts ist wieder ein Viereck, aber dieses Viereck besitzt keine Abschnitte. Am wichtigsten ist der obere Teil bei der Angabe eines Objekts, welcher unterstrichen wird. Unterstrichen wird die Angabe eines Objektname, gefolgt von einem Doppelpunkt :, gefolgt von dem Namen der Klasse von der dieses Objekt eine Instanz ist.

Schauen wir uns unser kleines Beispiel an. Sie können beide Angaben machen, also den Namen für das Objekt, z.B. „martin“, gefolgt von Doppelpunkt :, und dann die zugehörige Klasse „Benutzer“. Sie könnten aber auch das Objekt **anonym** halten, also keinen Objektname angeben und nur den Klassennamen angeben, von dem dieses Objekt eine Instanz ist. Wichtig bei der Verwendung von anonymen Objekten ist, dass Sie zuerst den Doppelpunkt : hinschreiben und dahinter den Namen der Klasse. Dies wird benötigt, um zu erkennen, dass es sich um den Klassennamen handelt und nicht um den Objektname. Also man würde dann schreiben „:Benutzer“. Sie könnten aber auch nur einen Objektname hinschreiben und keinen Klassennamen angeben. Diese Variante besteht jedoch eher theoretisch und ist in der Praxis nicht so sinnvoll weil Sie ja festhalten möchten, von welcher Klasse ein Objekt eine Instanz ist.

Im Anschluss an den unterstrichenen Bereich gebe ich die **Attribute mit den entsprechenden Werten** an, wie sie auch in der Klasse definiert sind. Die Werte liegen dann im Wertebereich der für das Attribut definiert ist. Der Wert „Martin“ des Attributs „name“ ist offensichtlich ein String, so wie es in der Klassendefinition des Attributs verlangt wird.

Ich mache beim Objekt üblicherweise keine Angaben von Operationen, weil dies ist überflüssig, weil sowieso alle Instanzen der Klasse dieselben Operationen haben. Ich gebe beim Objekt nur die Attribute mit ihren möglichen Werten an.

Objektdiagramm: Basiskonzepte

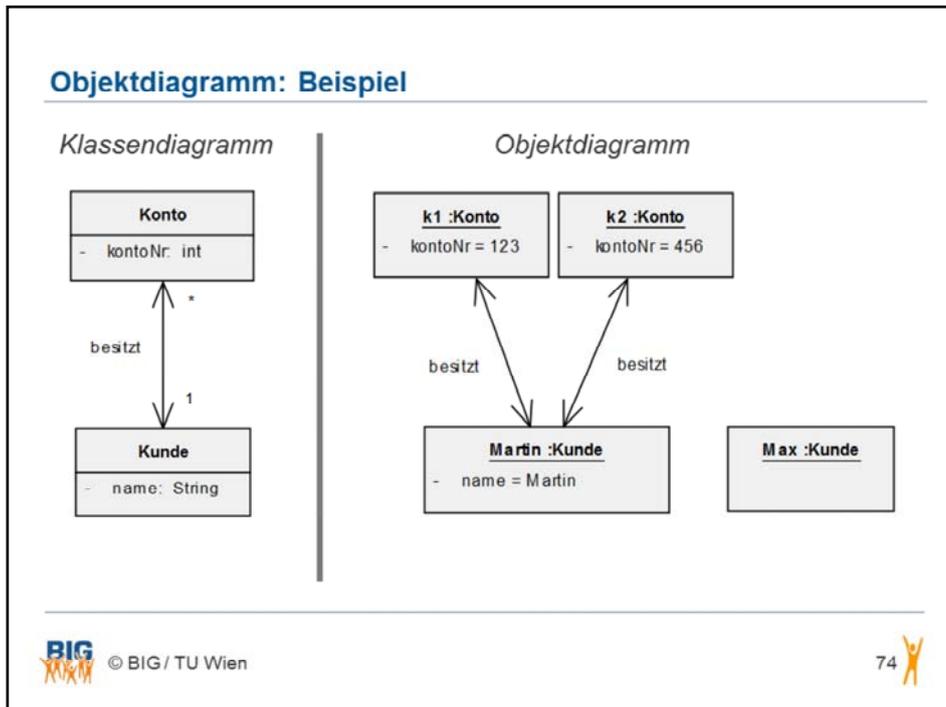
- **Basiskonzepte** des Objektdiagramms
 - Instanz einer Klasse: **Objekt**
 - Instanz einer Assoziation: **Link**
 - Instanz eines Datentyps: **Wert**
- **Einheitliche Notationskonventionen**
 - Gleiches Notationselement wie auf Typebene benutzen
 - Unterstreichen (bei Links optional)
- Objektdiagramm muss **nicht vollständig** sein
 - Z.B. können Werte benötigter Attribute fehlen, aber auch Instanzspezifikation abstrakter Klassen modelliert werden

Wir kommen kurz zur **Terminologie** im Objektdiagramm.

Wir sprechen bei der Instanz einer Klasse von einem **Objekt**, bei der Instanz einer Beziehung oder Assoziation von einem **Link** und bei der Instanz eines Datentyps sprechen wir von einem **Wert**.

Das Objektdiagramm muss nicht vollständig sein. Niemand zeichnet ein vollständiges Objektdiagramm. Die Modellierung aller Objekte die sich in einem System befinden erscheint nicht sinnvoll. Das Objektdiagramm wird eher exemplarisch verwendet. Beispielsweise wird es verwendet, um die Multiplizitäten, die ich in einem Klassendiagramm verwende, zu veranschaulichen.

Objektdiagramm: Beispiel



Wir haben hier auf der linken Seite ein Klassendiagramm mit den Klassen „Konto“ und „Kunde“ und eine Beziehung dazwischen. Ein Konto wird von genau einem Kunden besessen und ein Kunde kann 0 oder mehrere Konten besitzen. Dann wäre das rechte Bild die gültige Darstellung in einem Objektdiagramm. Ich habe zwei Objekte von „Konto“ und zwei Objekte von „Kunde“ und Links zwischen diesen Objekten.

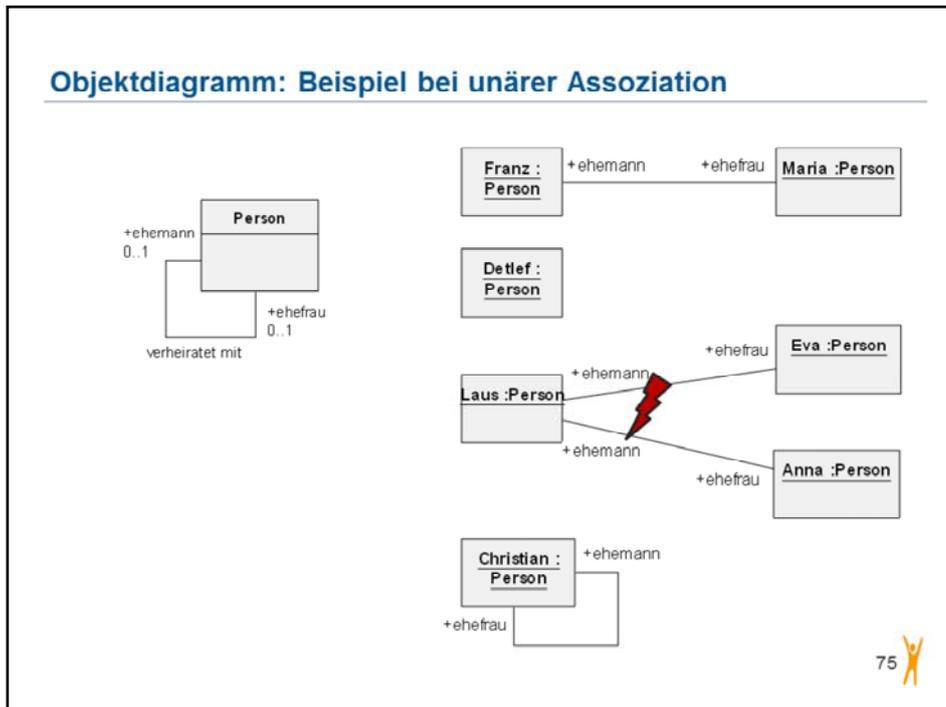
Es sei darauf hingewiesen, dass es im Objektdiagramm keine Multiplizitäten mehr gibt. Ich sehe nur, mit wie vielen Objekten jedes andere Objekt verbunden ist, um dadurch die Multiplizität feststellen zu können.

Jedes Konto in unserem Objektdiagramm wird genau von einem Kunden besessen. Konto „k1“ wird vom Kunden „Martin“ besessen und Konto „k2“ wird ebenfalls von „Martin“ besessen. Daher stimmt die Angabe „1“ als Multiplizität auf der Kunden-Seite im Klassendiagramm.

Umgekehrt, „Martin“ besitzt zwei Konten „k1“ und „k2“, „Max“ besitzt kein einziges Konto. Daher stimmt die Aussage, ein Kunde kann 0 bis mehr Konten besitzen und die Angabe eines Sterns * auf Seite des Kontos ist im Klassendiagramm richtig.

Was ich im Beispiel unseres Objektdiagramms nicht machen dürfte, hier zusätzlich einen Link zwischen dem Konto „k2“ und „Max“ aufnehmen. Denn dann wäre das Klassendiagramm nicht mehr gültig. Denn in diesem Falle würde dann das Konto „k2“ von zwei Kunden besessen werden, nämlich von „Martin“ und „Max“, jedoch im Klassendiagramm steht, dass ein Konto nur von einem Kunden besessen werden kann. Und genau dazu verwende ich oft das Objektdiagramm, um jemanden zu erklären: Du schau einmal, du hast dich da getäuscht, weil da ist in unserem Beispiel schon möglich, dass du eine zweite ausgehende „besitzt“-Beziehung von einem Konto hast. Wenn ich sage, ein Konto kann geteilt werden, es kann von mehreren Personen besessen werden. Für diesen Zweck wird das Objektdiagramm manchmal eingesetzt, um diese Multiplizitäten zu verdeutlichen. Ansonsten ist es eher ein gering verbreitetes Diagramm.

Objektdiagramm: Beispiel bei unärer Assoziation



Wir kommen zurück zum Beispiel einer Beziehung „ist verheiratet“, die als **unäre Beziehung** von und zur Klasse „Person“ dargestellt wird. Zusätzlich habe ich jedoch nun auf der rechten Seite auch beispielhaft ein Objektdiagramm, das diese Beziehung veranschaulichen soll.

Jetzt habe ich beispielsweise meinen „Franz“. „Franz“ ist eine Instanz von „Person“. Und ich habe „Maria“, die ebenfalls eine Instanz von „Person“ ist. Und die beiden sind miteinander verheiratet.

Es kann dann noch den „Detlef“ geben. „Detlef“ hat noch keine Frau gefunden und hat dementsprechend keine Beziehung zu jemand anderem, weil eine „0“ auch erlaubt ist.

Was nicht passieren darf ist der „Laus“, der die Person „Eva“ und die Person „Anna“ heiraten möchte, weil das wäre eine Verletzung. Ich hätte zwei ausgehende Beziehungen zu einer anderen Person in der Rolle „Frau“ und das Maximum ist „1“. Daher ist das natürlich nicht erlaubt und unser „Laus“ darf nicht zweimal heiraten.

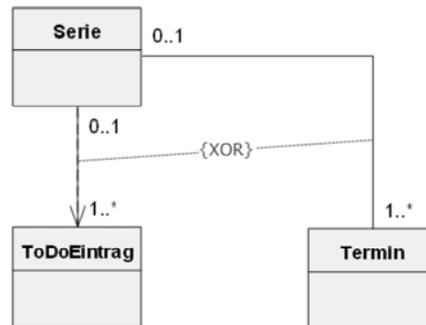
Und jetzt kommt der liebe „Christian“ und der ist komplett selbstverliebt und heiratet sich selbst. Die Frage, die sich uns nun stellt, ist dieses Objektdiagramm in dem sich „Christian“ selbst heiraten kann eine Verletzung von unserem Klassendiagramm auf der linken Seite? Die Antwort lautet nein. Die Darstellung im Objektdiagramm ist keine Verletzung, weil „Christian“ mit 0 bis 1 anderen Personen in Beziehung steht, halt einmal als „Mann“ und auch einmal als „Frau“, was semantisch nicht sehr sinnvoll ist oder möglich ist, aber syntaktisch wäre es noch korrekt. Um der Semantik gerecht zu werden, müsste man noch zusätzliche Einschränkungen, sogenannte Constraints formulieren. Eine richtige Einschränkung für unser Beispiel wäre, dass für jede Beziehung von „ist verheiratet“ auf der Objekt-Ebene gelten muss, dass das Quellobjekt und das Zielobjekt nicht das gleiche Objekt sein darf. D.h. die umgangssprachliche Aussage „Ich darf mich nicht selbst heiraten“ bedeutet, dass bei der Beziehung „ist verheiratet“ das Quellobjekt nicht gleich dem Zielobjekt sein darf. Das kann man graphisch nicht notieren und das sind Einschränkungen, die ich mit Hilfe der Object Constraint Language, oder kurz OCL, ausdrücken kann. Doch die OCL ist nicht Gegenstand dieser Lehrveranstaltung, sondern OCL wird in der Lehrveranstaltung Model Engineering im Masterstudium durchgenommen.

Inhalt

- Klassendiagramm
 - Klassen
 - Attribute und Operationen
 - **Assoziationen**
 - Schwache Aggregation
 - Starke Aggregation
 - Generalisierung
 - Zusammenfassendes Beispiel
 - Übersetzung nach Java
 - Datentypen in UML
- Objektdiagramm
- Paketdiagramm
- Abhängigkeiten

Exklusive Assoziation

- Für ein bestimmtes Objekt kann zu einem bestimmten Zeitpunkt **nur eine von verschiedenen möglichen Assoziationen** instanziiert werden: **{xor}**



Wir kehren zurück zu den Folien des Klassendiagramms.

Es gibt noch die Möglichkeit einer **exklusiven Assoziation**.

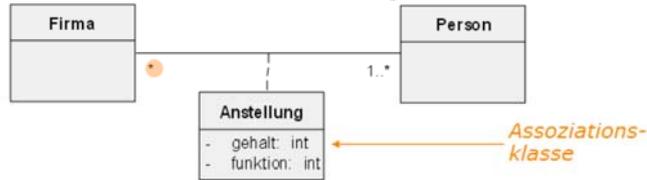
Ich habe hier in dem Beispiel eine Serie. Eine Serie beinhaltet entweder einen bis mehrere ToDo-Einträge oder einen bis mehrere Termine. Aber nur entweder oder.

Dementsprechend kann ich diese beiden Beziehungen gegeneinander ausschließen. Dies geschieht durch die strichlierte Linie zwischen den Assoziationen und durch die Angabe des XOR-Merkmals. Dementsprechend kann es sein, dass eine Serie, obwohl die Kardinalität auf 1 gesetzt ist, keinen ToDo-Eintrag enthält, weil ich zusätzlich noch die XOR-Variante bei der Beziehung berücksichtigen muss. D.h. ich habe mindestens einen Termin oder mindestens einen ToDo-Eintrag, aber die beiden sind „exklusiv oder“, durch eben das XOR ausgeschlossen. D.h. die Serie steht entweder mit einem bis mehreren ToDo-Einträgen oder mit einem bis mehreren Terminen in Beziehung. Eine Serie kann dementsprechend auch niemals gleichzeitig ToDo Einträge und Termine enthalten.

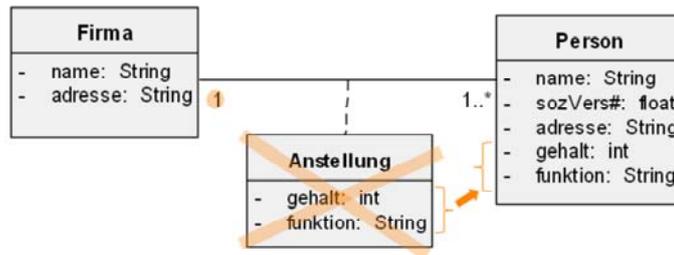
Assoziationsklasse (1/2)

▪ Kann Attribute der Assoziation enthalten

- Bei m:n-Assoziationen mit Attributen notwendig



- Bei 1:1 und 1:n-Assoziationen sinnvoll aus Flexibilitätsgründen (Änderung der Multiplizität möglich)



20

Wir kommen zur **Assoziationsklasse**. Die Assoziationsklasse dient dazu, Merkmale der Beziehung festzuhalten. Es sollte sich dabei wirklich um Merkmale der Beziehung handeln und nicht um Merkmale der beteiligten Klassen.

In unserem Fall haben wir eine Beziehung zwischen der Klasse „Firma“ und der Klasse „Person“ und ich möchte das „Gehalt“ und die „Funktion“ in der Assoziationsklasse festhalten.

Jetzt könnte ich fragen, warum gebe ich „Gehalt“ und „Funktion“ nicht als Attribute zu „Person“ bzw. als Attribute zur Firma? Zur Person kann ich „Gehalt“ und „Funktion“ nicht dazugeben, außer vielleicht das „Gehalt“, wenn ich die Gehaltssumme, die die Person über alle Firmen hinweg verdient, meine. Wenn ich das „Gehalt“ und die „Funktion“ bei der „Person“ hinzugebe, dann würde das bedeuten, dass diese Person in jeder Firma gleich viel verdient und in jeder Firma dieselbe Funktion hätte, was nicht der Fall sein wird. Umgekehrt kann ich „Gehalt“ und „Funktion“ nicht bei „Firma“ dazugeben, weil dann hätten alle Mitarbeiter dieser Firma die gleiche Rolle und würde das gleiche verdienen.

Das bedeutet, dass „Gehalt“ und „Funktion“ Merkmale der Beziehung sind, d.h. ich habe eine Sonderform einer Klasse, die Assoziationsklasse, die ich an die Beziehung dazu hängen.

Für eine Assoziationsklasse müssen wir auch einen Klassennamen vergeben. In unserem Fall verwenden wir den Klassennamen „Anstellung“. Im Rahmen dieser „Anstellung“, hat diese „Person“ bei dieser „Firma“ ein bestimmtes „Gehalt“ und eine bestimmte „Funktion“.

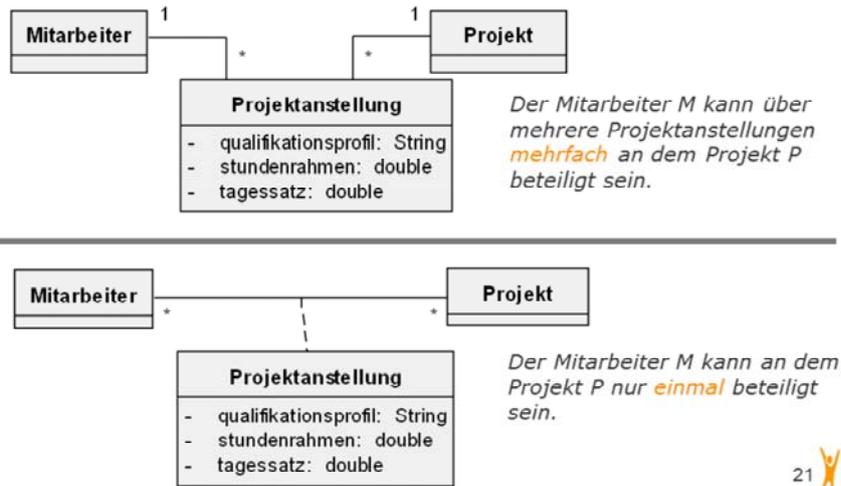
Die Assoziationsklasse ist notwendig für die Angabe von Merkmalen der Beziehung bei sogenannten n:m-Beziehungen. Eine Firma beschäftigt eine bis mehrere Personen, d.h. auf der Person-Seite habe ich gedanklich ein n stehen und eine Person ist bei m Firmen beschäftigt, dementsprechend handelt es sich um eine n:m-Beziehung.

Man könnte das obere Beispiel abändern und sagen, dass die Leute nicht mehr so viel arbeiten sollen. Wir wollen, dass jeder nur mehr in einer Firma arbeiten darf. Dann hätte ich auf der linken Seite auf einmal einen Einser, eine Person darf dann nur mehr in einer Firma arbeiten. Wenn ich eine 1:n-Beziehung habe, dann kann ich auch die Assoziations-Klasse auflösen, ob es semantisch sinnvoll ist, ist nicht ganz sicher. Auflösen kann ich es aber immer wie folgt: Ich kann die Attribute der Assoziations-Klasse auf der n-Seite hinzufügen, die n-Seite ist in unserem Fall die Person-Seite. „Gehalt“ und „Funktion“ wandern dann zur Klasse „Person“. Dies ist korrekt, weil diese „Person“ dann als Merkmal genau ein „Gehalt“ und eine „Funktion“ hat und sie ja nur noch in einer „Firma“ beschäftigt sein kann.

Bei einer Assoziationsklasse, im Falle von 1:1- oder 1:n-Beziehungen, kann die Assoziationsklasse aufgelöst werden. Bei 1:n-Beziehungen werden die Attribute immer zur n-Seite dazu geschrieben. Bei 1:1-Beziehungen kann ich mir das syntaktisch gesehen aussuchen. Meistens ist es semantisch gesehen klar, ob es auf die eine Seite wandert, oder auf die andere Seite. Wenn das untere Beispiel eine 1:1 Beziehung hätte, dann würde das bedeuten, dass jede Firmen nur mehr eine Person beschäftigen dürfen. Dann könnte ich es mir aussuchen, ob ich das Gehalt zur Firma oder zur Person hinzufüge. Ich persönlich würde es eher zur Person geben, aber ich könnte das Gehalt natürlich auch zur Firma geben, weil alle meine Angestellten, von denen es halt leider nur einen gibt, verdienen ja dann das Gleiche.

Assoziationsklasse (2/2)

- Normale Klasse ungleich Assoziationsklasse



Wir kommen noch zur Vertiefung der Assoziationsklasse.

Man könnte die Assoziationsklasse auch auflösen, indem man einfach eine ganz normale Klasse schafft. Hier haben wir die Klassen „Mitarbeiter“ und „Projekt“. Ich könnte die Merkmale „qualifikationsprofil“, „stundenrahmen“ und „tagessatz“ als Assoziationsklasse modellieren, wie im unteren Beispiel angegeben. Oder ich könnte wie im oberen Beispiel diese Assoziationsklasse auflösen und zwei Beziehungen erzeugen. Das ist semantisch ähnlich, aber nicht gleich.

Die Variante im oberen Beispiel bedeutet: Ein Mitarbeiter kann mehrere Projektanstellungen haben. Jede Projektanstellung bezieht sich auf genau ein Projekt. Umgekehrt gilt: in einem Projekt gibt es mehrere Projektanstellungen und jede Projektanstellung bezieht sich auf einen Mitarbeiter. Der wirkliche Unterschied ist: Der Mitarbeiter kann im oberen Beispiel die Projektanstellung „A“ haben und die ist dem Projekt „X“ zugeordnet. Der Mitarbeiter kann aber mehrere Projektanstellungen haben, er kann nicht nur die Projektanstellung „A“ haben, er kann auch noch die Projektanstellung „B“ haben, und die ist ebenfalls dem Projekt „X“ zugeordnet. Das bedeutet, dass ein Mitarbeiter über mehrere Projektanstellungen mehrfach an dem Projekt „X“ beteiligt sein kann.

Im unteren Beispiel gilt, dass ein Mitarbeiter zwar an mehreren Projekten beteiligt sein kann und ein Projekt mehrere Mitarbeiter beschäftigen kann, jedoch wird bei den Beziehungen die Eigenschaft „unique“ unterstellt, wenn nicht anders angegeben. Wenn ich „unique“ unterstelle, dann bedeutet das auf Objekt-Ebene, dass es in diesem Beispiel nur einen einzigen Link zwischen dem Mitarbeiter „Max“ und dem Projekt „X“ geben kann. Wenn ich unseren „Max“ im Projekt „X“ als Analyst einsetze, dann kann ich nicht noch einen weiteren Link zwischen „Max“ und „X“ aufnehmen und „Max“ dann in diesem Projekt als Programmierer einsetzen.

n-äre Assoziation (1/3)

- **Beziehung zwischen mehr als zwei Klassen**
 - Navigationsrichtung kann nicht angegeben werden
 - Multiplizitäten geben an, wie viele Objekte einer Rolle/Klasse einem festen (n-1)-Tupel von Objekten der anderen Rollen/Klassen zugeordnet sein können
- **Multiplizitäten implizieren Einschränkungen**, in einem bestimmten Fall funktionale Abhängigkeiten
- Liegen (n-1) Klassen (z.B. A und B) einer Klasse (z.B. C) mit Multiplizität von max. 1 gegenüber, so existiert eine funktionale Abhängigkeit $(A, B) \rightarrow (C)$

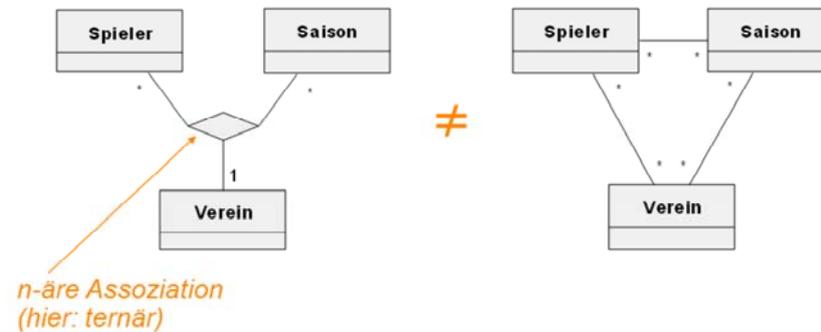
Wir kommen zu den **n-ären-Assoziationen**.

Die n-ären-Assoziationen sind jene Assoziationen, an denen drei oder mehrere Klassen beteiligt sind. Sind es genau 3, dann spricht man beispielsweise von ternären Assoziationen. n-äre Assoziationen kommen in der Praxis in einem Domänen-Modell noch vor. In der Implementierung verzichtet man aber meistens darauf und man verwendet nur binäre Assoziationen. Weil man muss sich fragen, wie wird denn eine ternäre Assoziation in ein objektorientiertes System umgelegt? Da geht man meistens her, dass man dann die ternäre Assoziation durch eine eigene Klasse darstellt und diese verbindet man mit binären Assoziationen zu jenen Klassen, die an der ursprünglichen ternären Beziehung beteiligt waren. Wie nun ternäre Assoziationen funktionieren und wie man die Multiplizitäten feststellt, sehen wir in der Folge am besten an einem Beispiel.

n-äre Assoziation (2/3)

▪ Beispiel

- (Spieler, Saison) → (Verein)
 - Ein Spieler spielt in einer Saison bei genau **einem** Verein
- (Saison, Verein) → (Spieler)
 - In einer Saison spielen bei einem Verein **mehrere** Spieler
- (Verein, Spieler) → (Saison)
 - Ein Spieler spielt in einem Verein in **mehreren** Saisonen



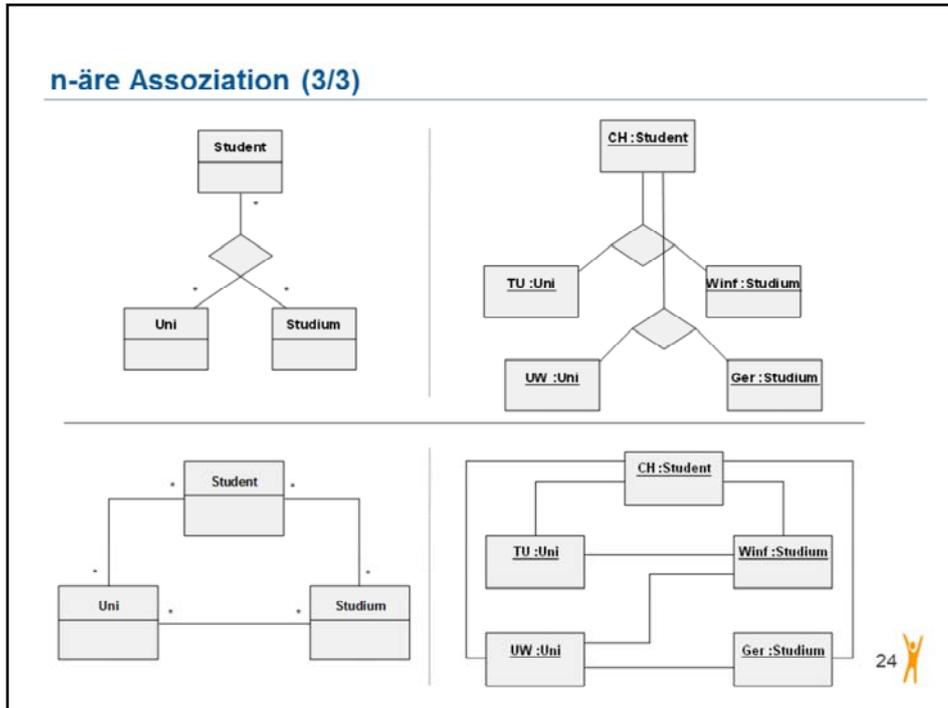
23

Wir haben hier ein Beispiel, das ich näher erklären möchte. Wir haben ein Informationssystem von einer sehr strikten Fußballliga. Es gibt Spieler, Vereine und Saisonen. Und ich nehme folgendes an: In meiner Liga dürfen die Spieler während der Saison nicht wechseln. Und jetzt geht es darum, die **Multiplizitäten** hier zu beschriften. Wie gehe ich dabei vor? Ich erweitere den Sonderfall einer binären Beziehung. Wie bin ich bei den binären Beziehungen vorgegangen? Ich habe den Satz immer begonnen mit „ein“ oder „eine“. Beispielsweise bei der Beziehung zwischen „Projekt“ und „Mitarbeiter“ sage ich: „In einem Projekt arbeiten wie viele Mitarbeiter?“ – „Mehrere“. Wenn ich die andere Seite beschriften möchte: „Ein Mitarbeiter arbeitet in wie vielen Projekten?“ – „In mehreren“. Wenn ich eine bestimmte Seite beschriften möchte, setze ich alle anderen Seiten auf 1 am Satzanfang. Eine binäre Assoziation ist eine Sonderform, weil es nur zwei beteiligte Klassen gibt und daher nur noch eine andere Klasse übrig bleibt. Wenn ich bei einer ternären Beziehung eine Seite beschriften möchte, muss ich alle anderen auf 1 setzen, d.h. die beiden anderen. Ich muss also fragen: „Ein Spieler spielt in einer Saison bei wie vielen Vereinen?“ - „Bei genau einem.“ Die Einschränkung war ja, dass meine Spieler während der Saison nicht wechseln dürfen. Dementsprechend habe ich bei der Klasse „Verein“ einen 1er. Umgekehrt, wenn ich die Seite der Spieler beschriften möchte, sage ich: „In einer Saison spielen bei einem Verein mehrere Spieler.“ Wenn ich die Saison beschriften möchte gilt: „Ein Spieler spielt bei einem Verein 0 bis mehrere Saisonen.“

Die Grundlage des Ganzen sind **funktionale Abhängigkeiten**. Diese werden Sie auch in der Datenmodellierung kennenlernen. Zur Bestimmung der Multiplizität von „Verein“ kann man auch fragen: „Gibt es, wenn ich den Spieler und die Saison kenne, eine eindeutige Antwort für den Verein?“ Wenn man das mit „Ja“ beantworten kann, schreibt man eine „1“ hin. Anders formuliert: „Spieler“ und „Saison“ miteinander bestimmen „Verein“. Das verhält sich wie bei Schlüsselattributen: Kenne ich den Spieler und kenne ich die Saison, dann gibt es nur einen Verein, bei dem er spielen kann. Umgekehrt gilt das aber nicht: In einer Saison und bei einem Verein kann ich nicht eindeutig einen Spieler nennen, weil es mehrere Spieler sind. Hier gilt die funktionale Abhängigkeit nicht, d.h. Saison und Verein bestimmen nicht den Spieler. Ähnlich verhält es sich, dass Verein und Spieler nicht die Saison bestimmen, weil ein Spieler bei einem bestimmten Verein ja auch mehrere Saisonen spielen kann.

Jetzt könnte man sich fragen, ob es nicht einfacher gewesen wäre, statt der ternären Beziehung drei binäre Beziehungen aufzunehmen. Wir haben da den Verein, wir haben den Spieler und wir haben die Saison. Ein Spieler spielt bei mehreren Vereinen, bei einem Verein gibt es mehrere Spieler. Ein Spieler spielt durchaus mehrere Saisonen, in einer Saison spielen durchaus mehrere Spieler. Ein Verein besteht auch in mehreren Saisonen, in einer Saison nehmen mehrere Vereine teil. Das ist das, was ich durch lauter binäre Assoziationen darstellen kann. Ich habe Ihnen aber zuvor einen Constraint genannt: Meine Spieler dürfen während der Saison nicht wechseln. Diese Information ist mir jetzt bei den binären Beziehungen durch den Rost gefallen. Weil der Fall, dass ein Spieler in einer Saison nur bei einem Verein spielen kann, wie durch den 1er auf der Seite des Vereins bei der ternären Beziehung dargestellt, ist eben eine zusätzliche Information, die mir im Fall von binären Beziehungen verloren geht. D.h. ich kann es durch die ternäre Beziehung graphisch darstellen und daher sieht man hier, dass die ternären Beziehungen durchaus Sinn machen.

n-äre Assoziation (3/3)



Ich sollte auch eine Beziehung zwischen Student, Universität und Studium mit einer ternären Beziehung modellieren. Jetzt werden wir draufkommen, dass es sich um eine n:m:p Beziehung handelt. Ein Student belegt an einer Uni mehrere Studien. An einer Uni wird ein Studium durchaus von mehreren Studenten betrieben. Ein Student studiert ein und dasselbe Studium unter Umständen auch an mehreren Unis.

Mittels binärer Assoziation würde ich das folgendermaßen modellieren: Ein Student studiert an mehreren Unis. An einer Uni studieren mehrere Studenten. Ein Student studiert mehrere Studien. Ein Studium wird von mehreren Studenten studiert. Ein Studium wird von mehreren Unis angeboten. An einer Uni gibt es mehrere Studien.

Das schaut ja ziemlich ähnlich aus. Sind die beiden Klassendiagramme tatsächlich äquivalent oder nicht? Sehen wir uns das an einem konkreten Beispiel, dargestellt mittels Objektdiagrammen auf der rechten Seite, an.

Nehmen wir an, das obere Objektdiagramm der ternären Beziehung spiegelt den tatsächlichen Sachverhalt wieder. Ich, der Student „CH“, studiere an der TU Wirtschaftsinformatik und ich als „CH“ studiere an der Uni Wien Germanistik. Dann müsste ich im unteren Objektdiagramm folgendes einzeichnen: Ich „CH“ studiere an der „TU“. Ich „CH“ studiere an der Uni Wien „UW“. Ich „CH“ studiere „Winf“. Und ich „CH“ studiere Germanistik „Ger“. „Winf“ wird an der „TU“ angeboten. Und „Winf“ wird an der Uni Wien „UW“ angeboten. Germanistik „Ger“ wird nur an der Uni Wien „UW“ angeboten. Wenn ich nun diese Informationen über die Objekte, die da entsteht, zusammenwürfle, würde das ein falsches Bild ergeben. Nämlich „CH“ studiert Wirtschaftsinformatik, Wirtschaftsinformatik wird an der TU angeboten und Wirtschaftsinformatik wird an der Uni Wien angeboten und da „CH“ an beiden Standorten studiert, wird er dazu verdammt, Wirtschaftsinformatik sowohl an der TU, als auch auf der Uni Wien zu studieren. Aber unser „CH“ hat sich ja entschieden, Wirtschaftsinformatik nur an der TU zu studieren weil es da viel besser ist. Dementsprechend entsteht hier ein falscher Informationsgehalt, der besser durch ternäre Beziehungen ausgedrückt ist.

Dieses Beispiel sollte die Anwendung von ternären Beziehungen verdeutlichen. Aber bitte gehen Sie auf keinem Fall jetzt her und bauen in Ihren Modellen überall ternäre Beziehungen ein. Die kommen nur ganz selten vor. Ich habe das Problem, dass wenn man die zu deutlich durchmacht, dass alle probieren immer und überall ternäre Beziehungen einzusetzen. Diese müssen Sie besonders rechtfertigen, dass Sie diese brauchen. Im Normalfall wird Ihr Modell Großteils aus binären Beziehungen bestehen.

Aggregation

- Aggregation ist eine spezielle Form der Assoziation mit folgenden Eigenschaften:
 - Transitivität:
C ist Teil von B u. B ist Teil von A \rightarrow C ist Teil von A
 - Bsp.: Kühlung = Teil von Motor & Motor = Teil von Auto
 \rightarrow Kühlung ist (indirekter) Teil von Auto
 - Anti-Symmetrie:
B ist Teil von A \rightarrow A ist nicht Teil von B
 - Bsp.: Motor ist Teil von Auto, Auto ist nicht Teil von Motor
- UML unterscheidet zwei Arten von Aggregationen:
 - Schwache Aggregation (shared aggregation)
 - Starke Aggregation – Komposition (composite aggregation)

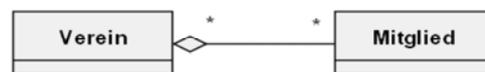
Wir kommen dann zum Konzept der **Aggregation**. Wobei wir bei der Aggregation zwischen zwei Arten unterscheiden: Die schwache Aggregation und die starke Aggregation.

Aggregation bedeutet eine „ist Teil von“-Beziehung. „Teil von“-Beziehung bedeutet, dass es ein übergeordnetes Ganzes gibt, das einen untergeordneten Teil beinhaltet. Wenn ich sagen kann: „C“ ist ein Teil von „B“ und „B“ ist ein Teil von „A“, dann ist, transitiv gesehen, „C“ auch ein Teil von „A“. Umgekehrt gilt bei Asymmetrie: wenn „B“ ein Teil von „A“ ist, kann „A“ nicht ein Teil von „B“ sein. Aufgrund der Transitivität braucht man gerichtete Graphen und die Antisymmetrie sagt aus, dass es keine Zyklen im Graph geben darf.

Schwache Aggregation



- Schwache Zugehörigkeit der Teile, d.h. Teile sind **unabhängig** von ihrem Ganzen
- Die Multiplizität des aggregierenden Endes der Beziehung (Raute) kann > 1 sein
- Es gilt nur eingeschränkte **Propagierungssemantik**
- Die zusammengesetzten Objekte bilden einen **gerichteten, azyklischen Graphen**



Die **schwache Aggregation** wird durch eine weiße Raute auf Seiten des Ganzen dargestellt.

Die schwache Aggregation wird oft als syntactic sugar bezeichnet. Das bedeutet, dass es zwar nett ist, sie angeben zu können, dass sie sich letztendlich aber nicht auf den Code durchschlägt. Die schwache Aggregation hat nur den Zweck eine Ganze-Teil-Beziehung zu kennzeichnen. Aber es besteht eine **schwache Zugehörigkeit der Teile zum Ganzen**. Das bedeutet, die Teile gehen nicht im Ganzen unter. Die **Teile sind unabhängig vom Ganzen**.

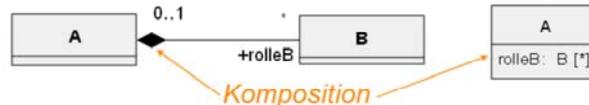
Das bedeutet, ein Verein, das Ganze, besteht aus seinen Mitgliedern, den Teilen. Die Mitglieder gehen aber nicht im Verein unter, d.h. sie können selbstständig bestehen. Dementsprechend kann ein Teil auch in mehreren Ganzen enthalten sein. Darum kann die **Multiplizität auf Seiten des Ganzen**, dort wo sich die weiße Raute befindet, auch größer als 1 sein. Ein Mitglied kann in mehreren Vereinen Mitglied sein. Sie können im Turnverein Mitglied sein, Sie können im Sportverein Mitglied sein usw.

Bei der schwachen Aggregation gilt auch nur eine **eingeschränkte Propagierungssemantik**. Was bedeutet das? Schauen wir uns das beim Löschen an. Propagieren würde bedeuten, wenn ich das Ganze lösche, dann lösche ich auch den Teil. Doch das darf bei der schwachen Aggregation nicht sein. Wenn ich den Verein lösche, so lösche ich noch lange nicht die Mitglieder des Vereins. Die Mitglieder können weiter bestehen.

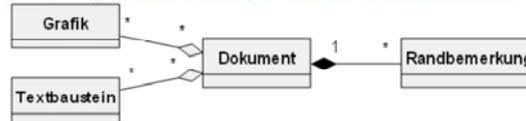
Wie bereits auf der vorigen Folie erwähnt, bilden die durch schwache Aggregation verbundenen Objekte einen **gerichteten, azyklischen Graphen**.

Da es sich bei der schwachen Aggregation um syntactic sugar handelt, kann man letztlich immer argumentieren ob es reicht, eine normale Assoziation zu verwenden, oder ob man eine schwache Aggregation verwenden sollte. Das kann man diskutieren. Wichtig ist bei der Übungsstunde, dass Sie argumentieren können, warum Sie eine schwache Aggregation oder eine normale Assoziation gewählt haben.

Starke Aggregation (= Komposition)



- Ein bestimmter Teil darf zu einem bestimmten Zeitpunkt in **maximal einem zusammengesetzten Objekt enthalten** sein
- Die **Multiplizität** des aggregierenden Endes der Beziehung kann (maximal) 1 sein
- **Abhängigkeit** der Teile vom zusammengesetzten Objekt
- **Propagierungssemantik**
- Die zusammengesetzten Objekte bilden einen **Baum**



Im Unterschied dazu gibt es auch die **starke Aggregation**, bei der der Teil im Ganzen untergeht. Es besteht eine **starke Abhängigkeit der Teile vom zusammengesetzten Objekt**. Wir haben hier die **Propagierungssemantik**. Das bedeutet, wenn ich das Ganze lösche, so werden auch seine Teile gelöscht.

Hier ist in dem abstrakten Beispiel „B“ ein Teil von „A“, in Form einer starken Aggregation. Das bedeutet, der Teil „B“ geht im Ganzen „A“ unter. Daher kann der Teil „B“ nicht in mehrere Ganzen „A“ eingebaut werden, weil er ja in einem Ganzen untergeht.

Die Raute befindet sich auf der Seite des Ganzen. Bei der schwachen Aggregation ist sie weiß, bei der starken Aggregation ist sie gefüllt, oder schwarz. Wenn ein Teil nicht in mehreren Ganzen eingebaut werden kann, dann kann das Maximum auf der Seiten der schwarzen Raute folglich immer nur ein 1er sein.

Es gibt hier unterschiedliche Diskussionen ob es immer ein 1er sein muss, oder ob 0 bis 1 auch erlaubt sein kann. Wir unterrichten, dass 0 bis 1 auch erlaubt sein kann. In manche Bücher haben Sie eine strengere Forderung an die „Existenzabhängigkeit“. Das bedeutet ein Teil kann nicht ohne seinem Ganzen existieren. Das wird von manchen gefordert. Aus praktischen Überlegungen verzichten wir auf diese Forderung.

Ein Beispiel: Ich erzeuge einen Motor und der ist für sich selbst existenzfähig. Sobald ich diesen Motor, der über seine Motorenummer eindeutig identifizierbar ist, in ein Auto einbaue, geht der Motor in Auto unter. D.h. ich kann denselben Motor nicht in ein zweites Auto einbauen. Nun fahre ich mit dem Auto gegen einen Baum - Totalschaden. Wir löschen ein Auto beim Totalschaden. Auf Grund der starken Aggregation wird dann der Motor im dem Auto auch gelöscht. Ich hätte aber noch die Möglichkeit, wenn ich mir meinen Totalschaden anschau, und mir denke „Alles hin, nur der Motor könnte vielleicht noch funktionieren“, dann hätte ich die Möglichkeit, vor dem Löschen des Autos den Motor noch auszubauen und für sich selbst wieder lebensfähig zu machen.

Manche Bücher schreiben, dass der Motor nicht für sich alleine existieren kann, sondern nur, wenn er in einem Auto eingebaut ist. Die Kardinalität 0 bis 1 ist die Interpretation, die ich zuvor vorgeführt habe: Ich kann einen Motor zuerst selbst haben und ich kann ihn aber dann auch explizit wieder ausbauen aus dem Ganzen, um ihn lebensfähig zu erhalten. Daher ist hier 0 bis 1 erlaubt.

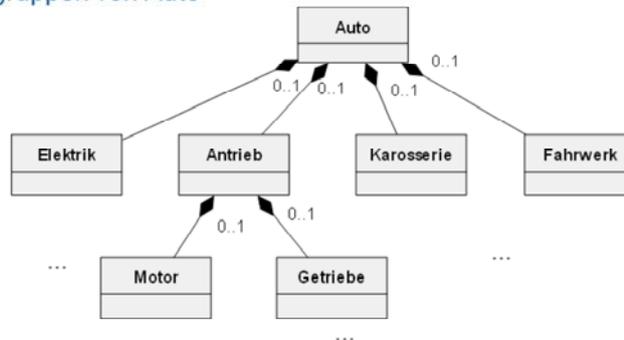
Die **Multiplizität ist immer maximal 1**. Die zusammengesetzten Objekte bilden einen **Baum**, weil ich ja den Teil nicht in einem zweiten Ganzen einbauen kann.

Noch ein gutes Beispiel: Ich habe ein Dokument und in diesem Dokument kann ich Kommentare einfügen. Diese Kommentare sind mit einer starken Aggregation an das Dokument gebunden. Wenn ich das Dokument lösche, dann gehen auch diese Kommentare, die jemand eingefügt hat, verloren. Ein Dokument beinhaltet 0 oder mehrere Randbemerkungen. Eine Randbemerkung ist maximal in einem Dokument vorhanden – Höchstzahl von der schwarzen Raute erreicht. In dem Fall macht 0 bis 1 bei der schwarzen Raute auch keinen Sinn, weil ich eine Randbemerkung nicht ins Leere schreiben kann. Ich muss sie bei einem Dokument hinzufügen.

Umgekehrt habe ich Grafiken oder Textbausteine. Ich kann ja eine Grafik, die als separate Datei vorliegt, in meinem Dokument per Referenz einbinden, d.h. wenn sich das File ändert, ändert sich auch die Grafik in meinem Dokument. Ich füge sie nicht mit Copy-und-Paste ein, sondern mittels einer Referenz. Dass wäre dann der Fall einer schwachen Aggregation. Weil wenn ich das Dokument lösche, so wird die Graphik immerhin noch weiter bestehen.

Starke Aggregation

- Mittels starker Aggregation kann eine Hierarchie von „Teil-von“-Beziehungen dargestellt werden (Transitivität!)
- Beispiel: Baugruppen von Auto



Wir haben hier noch das Beispiel von einem Auto.

Man sieht schön, dass mittels starker Aggregation eine Hierarchie von „Teil-von“-Beziehungen dargestellt werden kann und dass der resultierende Graph einen Baum darstellt. Das ist ein typisches Beispiel wie es auch in vielen Büchern vorkommen.

Ein Auto besteht aus einer Elektrik, einem Antrieb, einer Karosserie und einem Fahrwerk. Das ganze geht rekursiv weiter. Der Antrieb besteht aus einem Motor und aus einem Getriebe. In unserem Beispiel sind die Teile für sich selbst existenzfähig und daher haben wir immer auf Seiten der schwarzen Raute ein 0 bis 1.

Starke Aggregation vs. Assoziation - Faustregeln

- **Einbettung**
 - Die Teile sind i.A. physisch im Kompositum enthalten
 - Über Assoziation verbundene Objekte werden über Referenzen realisiert
- **Sichtbarkeit**
 - Ein Teil ist nur für das Kompositum sichtbar
 - Das über eine Assoziation verbundene Objekt ist i.A. öffentlich sichtbar
- **Lebensdauer**
 - Das Kompositum erzeugt und löscht seine Teile
 - Keine Existenzabhängigkeit zwischen assoziierten Objekten
- **Kopien**
 - Kompositum und Teile werden kopiert
 - Nur die Referenzen auf assoziierte Objekte werden kopiert

Auf dieser Folie sind die **Merkmale von starker und schwacher Aggregation** noch einmal zusammengefasst.

Man kann hier schön die Unterschiede sehen. Ich möchte diese Unterschiede noch einmal auch anhand unseres Dokuments mit Randbemerkungen und referenzierter Graphik erläutern.

Einbettung: Die Randbemerkung ist im Dokument eingebettet. Die Grafik wird über eine Referenz realisiert.

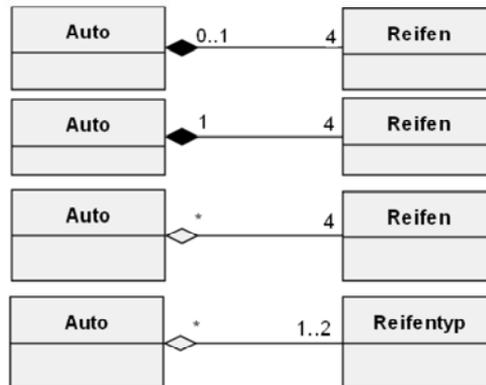
Sichtbarkeit: Die Randbemerkung ist nur für das Ganze sichtbar, während die Grafik auch als eigene Datei von außen sichtbar ist.

Die **Lebensdauer:** Das Ganze erzeugt und löscht seine Teile. Die Randbemerkung wird mit gelöscht, wenn ich das Dokument lösche, während die referenzierten Grafiken weiterhin bestehen bleiben, wenn ich das Dokument lösche.

Wenn ich eine **Kopie** mache, wird das ganze Dokument kopiert, d.h. auch die Randbemerkungen werden mit kopiert, während ich bei Referenzen nicht eine neue Grafik anlege, sondern nur die Referenzen auf diese Graphik kopiere.

Komposition und Aggregation

- Welche der folgenden Beziehungen trifft zu?



Hier noch einige Beispiele zum Verständnis.

Welche der folgenden Beziehungen treffen zu, oder welche sind richtig modelliert?

Im ersten Beispiel: Ein Auto hat 4 Reifen, ein Reifen gehört zu 0 bis einem Auto.

Oder zweites Beispiel: Ein Auto hat 4 Reifen, aber ein Reifen gehört zu genau einem Auto.

Zum dritten Beispiel: Ein Auto beinhaltet 4 Reifen, aber ein Reifen kann auf mehreren Autos montiert werden.

Oder viertes Beispiel: Einem Auto sind ein bis zwei Reifentypen zugeordnet und ein Reifentyp kann auf mehreren Autos montiert werden.

Komposition und Aggregation

- Welche der folgenden Beziehungen trifft zu?



Das erste Beispiel ist nach meiner Ansicht nach richtig: Ein Reifen kann auch ohne Auto existieren. Wir nehmen einmal einen Reifen und der kann auf einem Auto montiert sein oder auch nicht. Wichtig ist, wir reden hier wirklich von einem Reifen, einem physischen Reifen.

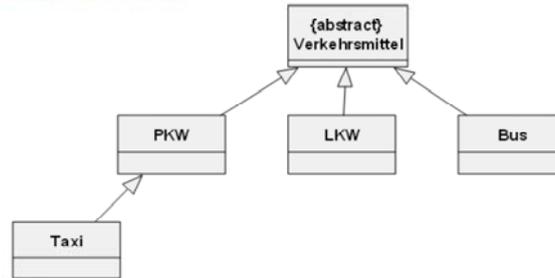
Zum zweiten Beispiel: Ein Reifen kann ohne Auto nicht existieren, d.h. ich nehme meinen Reifen als Informationssystem gar nicht auf, sondern erst dann, wenn dieser Reifen auf ein Auto aufgesteckt wird. Hier ist diese Lösung als falsch eingezeichnet, man könnte natürlich darüber diskutieren, wenn man das explizit so modellieren möchte, wie ich es gerade beschrieben habe.

Zum dritten Beispiel: Dass ein Reifen, ein physischer Reifen, auf mehrere Autos montiert wird, wie in der dritten Lösung, wird schwer gehen - zumindest nicht gleichzeitig. Daher ist diese Lösung sicherlich falsch.

Aber viel entscheidender ist die Frage im vierten Beispiel: Glauben wir wirklich, dass ich einen einzelnen physischen Reifen tracken werde oder rede ich nicht doch vom Reifentyp? Den Reifentyp „Semperit X10“ kann ich natürlich schon auf mehrere Autos montieren. Auf einem Auto darf ich nicht 4 Reifentypen montieren, weil auf den Achsen dieselben Reifentypen montiert sein sollten. Daher ist die Kardinalität 1 bis 2 im letzten Beispiel als richtig anzusehen. Nachdem ich nur zwei Achsen habe, kann ich maximal zwei unterschiedliche Reifentypen montieren.

Generalisierung

- **Taxonomische Beziehung** zwischen einer spezialisierten Klasse und einer allgemeineren Klasse
 - Die spezialisierte Klasse **erbt** die Eigenschaften der allgemeineren Klasse
 - Kann **weitere Eigenschaften** hinzufügen
 - Eine **Instanz der Unterklasse** kann überall dort verwendet werden, wo eine **Instanz der Oberklasse** erlaubt ist (zumindest syntaktisch)
- Mittels Generalisierung wird eine Hierarchie von „ist-ein“-Beziehungen dargestellt (Transitivität!)



32

Kommen wir zur **Generalisierung** oder Vererbung.

Vererbung wird durch Pfeile dargestellt, aber einer ganz besonderen Form von Pfeilen, wo die Spitze durch ein weißes, nicht gefülltes Dreieck dargestellt wird. Ich habe die Pfeilspitze immer bei der Superklasse und keine Pfeilspitze bei der Subklasse.

Die Generalisierung ist eine **taxonomische Beziehung** zwischen einer spezialisierten Klasse, der Subklasse, und einer allgemeineren Klasse, der Superklasse. Dadurch ergeben sich zu Englisch „is a“, oder zu Deutsch „ist ein“ Beziehungen.

In unserem Beispiel: Ein PKW ist eine besondere Form von einem Verkehrsmittel. Ein LKW ist eine besondere Form von einem Verkehrsmittel. Und auch der Bus ist eine besondere Form eines Verkehrsmittels.

Die spezialisierte Klasse, Subklasse, erbt die Eigenschaften der allgemeineren Klasse, der Superklasse. Geerbt werden insbesondere die Attribute, die Operationen, aber auch die Beziehungen zu anderen Klassen. In der Subklasse können dann natürliche weitere Eigenschaften hinzugefügt werden. Es ist natürlich insbesondere nur sinnvoll eine Subklasse aufzunehmen, wenn dann neue Attribute, neue Operationen oder neue Beziehungen hinzukommen, auch wenn das aus unserem Beispiel nicht unmittelbar ersichtlich ist.

Wichtig ist, dass aufgrund der „is a“ Beziehung, auch eine Instanz der Unterklasse überall dort verwendet werden kann, wo eine Instanz der Oberklasse erlaubt ist.

In unserem Beispiel sehen wir, dass Verkehrsmittel als abstrakt definiert ist. Zu den abstrakten Klassen kommen wir auf der nächsten Folie.

Abstrakte Klasse (1/2)

- Klasse, die **nicht instanziiert** werden kann
- **Nur in Generalisierungshierarchien** sinnvoll
- Dient zum "**Herausheben**" **gemeinsamer Merkmale** einer Reihe von Unterklassen
- Notation: Schlüsselwort {abstract} oder Klassenname in kursiver Schrift



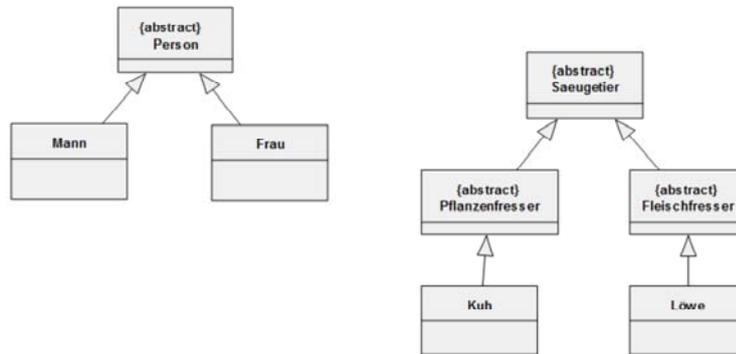
- Mit analoger Notation wird zwischen konkreten (= implementierten) und abstrakten (= nur spezifizierten) **Operationen** einer Klasse unterschieden

Ich kann Klassen als **abstrakt** definieren.

Abstrakt bedeutet, dass es keine Instanz der Superklasse geben kann. Es werden stattdessen die Subklassen instanziiert. Dementsprechend sind abstrakte Klassen nur in Generalisierungshierarchien sinnvoll, wo es dann von einer abstrakten Klasse dann auch nicht-abstrakte Subklassen gibt. Abstrakte Klassen dienen vor allem zum Herausheben gemeinsamer Merkmale einer Reihe von Unterklassen. So sollten beispielsweise zwei Unterklassen die selben Attribute haben, so gebe ich sie nicht in der Unterklasse an, sondern in einer neu geschaffenen abstrakten Superklasse. Das Schlüsselwort „abstract“ wird in geschwungenen Klammern {} in dem Abschnitt der Klasse hinzugefügt, steht hier über dem Klassennamen. Sie können das Schlüsselwort „abstract“ aber auch weg lassen und dafür den Klassennamen kursiv schreiben. Bei den Lösungen von Übungen oder Tests bitte immer das Schlüsselwort „abstract“ ausformuliert dazu schreiben.

Abstrakte Klasse (2/2)

- Beispiele:



Hier finden wir weitere Beispiele zu abstrakten Klassen.

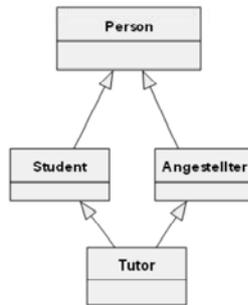
Im ersten Beispiel auf der linken Seite, kann es keine Person geben, die nicht Mann oder Frau ist, es müssen entweder instanziierte Männer oder Frauen sein.

Im zweiten Beispiel auf der rechten Seite sehen wir, dass man das Abstrahieren auch über mehrere Ebenen hinweg nach unten hin weiterführen kann. D.h. man kann auch abstrakte Klassen von abstrakten Klassen erben lassen. Wichtig ist, dass es sinnvollerweise dann am Ende der Vererbungshierarchie aber auch nicht-abstrakte Subklassen gibt.

Mehrfachvererbung

- Klassen müssen nicht nur eine Oberklasse haben, sondern können auch von mehreren Klassen erben

- Beispiel:



Mehrfachvererbung bedeutet, dass eine Subklasse zwei oder mehrere Superklassen besitzt. Mehrfachvererbung ist in UML erlaubt, das kann man modellieren. Die Frage jedoch ist, ob es sinnvoll ist, die Mehrfachvererbung einzusetzen. In einem Domänenmodell: Ja. In einem plattformspezifischen Modell, wenn ich schon weiß, welche Programmiersprache ich einsetze und wenn ich weiß, dass diese Programmiersprache Mehrfachvererbung nicht unterstützt, dann ist es nicht so schlau, wenn ich Mehrfachvererbung einsetze. Legen Sie hier die ganze Anwendung praktisch aus.

Generalisierung: Eigenschaften (1/2)

- Unterscheidung kann vorgenommen werden in
 - **Unvollständig / vollständig:**
In einer vollständigen Generalisierungshierarchie muss jede Instanz der Superklasse auch Instanz mindestens einer Subklasse sein
 - **Überlappend / disjunkt:**
In einer überlappenden Generalisierungshierarchie kann ein Objekt Instanz von mehr als einer Subklasse sein
 - **Default:** unvollständig, disjunkt



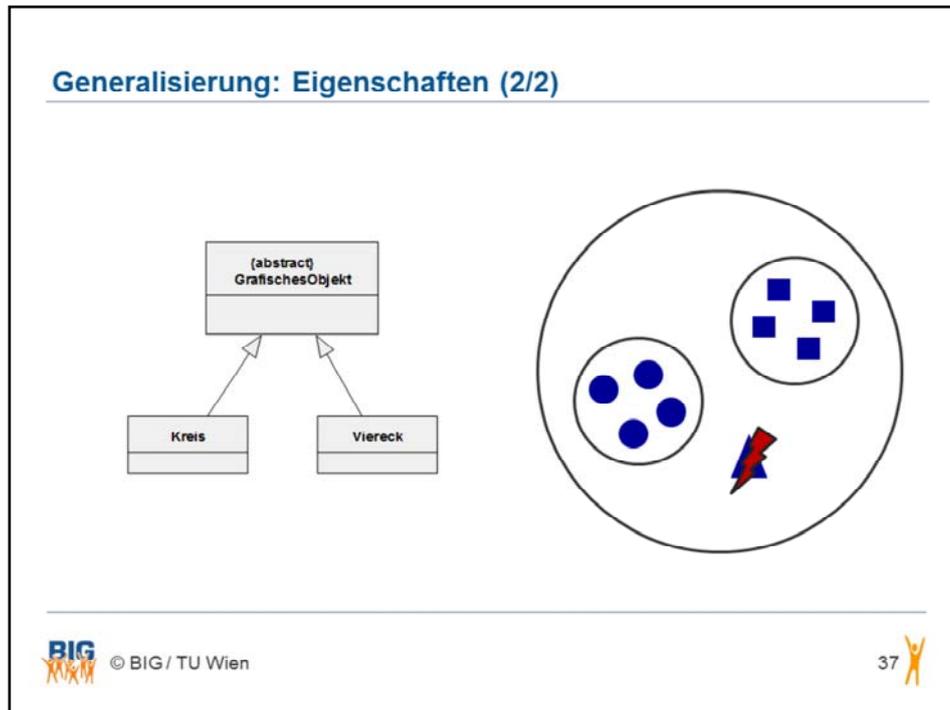
Wenn ich die Beziehung zwischen Super- und Subklassen in einer Generalisierungshierarchie betrachte, kann man sich auch die Frage stellen: Wie erfolgt die Spezialisierung in der Subklasse? Generalisierungshierarchien kann man anhand von folgenden Merkmalen unterscheiden: „**vollständig**“ bzw. „**unvollständig**“ und „**überlappend**“ bzw. „**disjunkt**“.

In einer **vollständigen** Generalisierungshierarchie muss jede Instanz der Superklasse auch eine Instanz mindestens einer Subklasse sein. Dies gilt für **unvollständige** Generalisierungshierarchien nicht.

In einer **überlappenden** Generalisierungshierarchie kann ein Objekt Instanz von mehr als einer Subklasse sein. In **disjunkten** Generalisierungshierarchien kann eine Instanz nur von einer Subklasse sein, aber nicht von zwei Subklassen.

Das sehen wir uns hier am Beispiel von „Eintrag“ an. Beim „Eintrag“ gibt es die Subklassen „Termin“, „Serie“ und „ToDoEintrag“. Es ist nicht möglich, dass ein Eintrag gleichzeitig eine Serie und ein Termin ist. Zur Frage ob es sich um eine unvollständige oder vollständige Aufteilung handelt: In diesem Beispiel ist die Superklasse abstrakt. D.h. es liegt nahe, dass es sich um eine vollständige Aufteilung handeln muss.

Generalisierung: Eigenschaften (2/2)



Am Schönsten sieht man diese Aufteilung anhand der Mengenlehre. Wenn man graphische Objekte hat, und zwar „Kreis“ und „Viereck“, dann gibt es die Menge der „graphischen Objekte“ und die Subklassenbildung ist nichts anderes als eine Teilmengenbildung, d.h. es gibt die Untermenge der Kreise und die Untermenge der Vierecke.

Es gibt also die Superklasse „Graphisches Objekt“ und diese vererbt einmal nach „Kreis“ und einmal nach „Viereck“. Das wäre mengenmäßig gesehen eine **vollständige Aufteilung** der Objekte.

Was bei gegebener Klassenhierarchie nicht möglich wäre ist ein graphisches Objekt „Dreieck“. Denn das wäre dann zwar ein „Graphisches Objekt“, aber dieses ist nicht in den Submengen „Kreis“ bzw. „Viereck“ enthalten. Es ergäbe sich also eine **unvollständige Aufteilung**.

Wenn die Klasse „Graphisches Objekt“ noch dazu als „abstrakt“ definiert ist, dann darf es ein „Dreieck“ natürlich gar nicht geben, weil es keine Instanzen von der abstrakten Superklasse „Graphisches Objekt“ geben darf sondern nur von „Kreis“ bzw. „Viereck“.

Generalisierung: Eigenschaften (1/2)

- Unterscheidung kann vorgenommen werden in
 - **Unvollständig / vollständig:**
In einer vollständigen Generalisierungshierarchie muss jede Instanz der Superklasse auch Instanz mindestens einer Subklasse sein
 - **Überlappend / disjunkt:**
In einer überlappenden Generalisierungshierarchie kann ein Objekt Instanz von mehr als einer Subklasse sein
 - **Default:** unvollständig, disjunkt



Wir kommen noch einmal auf die Beispiele auf der vorhergehenden Folie zurück. Im ersten Beispiel haben wir also eine **vollständige** Aufteilung, d.h. ein Objekt ist entweder ein „Termin“, eine „Serie“ oder ein „ToDoEintrag“. Es handelt sich sogar um eine „entweder-oder“ Beziehung: Ein Objekt ist entweder ein „Termin“ oder eine „Serie“ oder ein „ToDoEintrag“ und damit sind sie auch **disjunkt**. Umgekehrt gibt es im zweiten Beispiel die Klasse „Person“ und ihre beiden Subklassen, die Menge der „Angestellten“ und die Menge der „Studenten“. Jetzt gibt es aber Personen, die weder Angestellte, noch Studenten sind, d.h. es handelt sich dabei um direkte Einträge von der Superklasse „Person“ und daher um eine **unvollständige Aufteilung** in Englisch „**incomplete**“. Andererseits kann ein Objekt auch sowohl Angestellter sein, als auch Student, d.h. es gilt „**overlapping**“, „überlappend“.

Im Normalfall nimmt man an, dass Überlappung nicht gegeben ist. Überlappung ist auch ein Problem bei Programmiersprachen, da diese es üblicherweise nicht unterstützen. Und im Normalfall geht man auch noch davon aus, dass Vererbungen „incomplete“ sind, weil es normalerweise direkte Instanzen von Superklassen geben kann, außer es handelt sich um eine abstrakte Klasse, da geht man immer von einer vollständigen Aufteilung aus.

Generalisierung: Redefinition von geerbten Merkmalen (1/2)

- Geerbte Merkmale können **in Subklasse redefiniert** werden
 - {redefines <feature>}
- **Redefinierbare Merkmale**
 - Attribute
 - Navigierbare Assoziationsenden
 - Operationen
- Redefinition von Operationen in (in)direkten Subklassen kann auch verhindert werden, indem die Operation mit der Eigenschaft {leaf} gekennzeichnet wird
- Das redefinierte Merkmal muss konsistent zum ursprünglichen Merkmal sein – verschiedenste **Konsistenzregeln** – z.B.
 - Ein redefiniertes Attribut ist konsistent zum ursprünglichen Attribut, wenn sein **Typ gleich oder ein Subtyp** des ursprünglichen Typs ist
 - Das **Intervall der Multiplizität** muss in jenem des ursprünglichen Attributs **enthalten** sein
 - Die Signatur einer Operation muss die **gleiche Anzahl an Parametern** aufweisen, etc.

38 

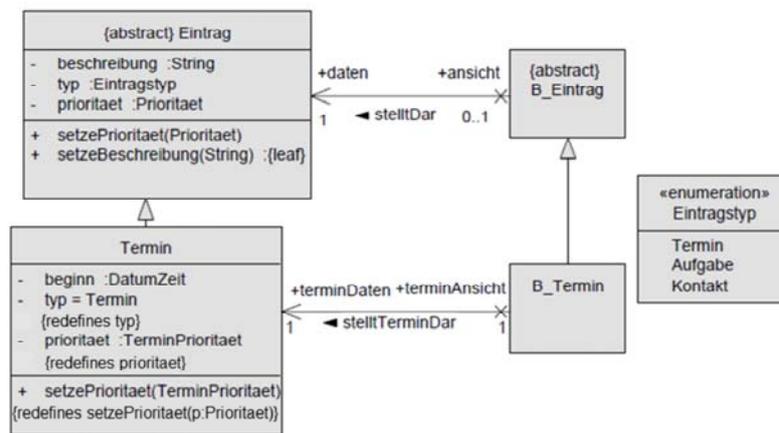
Bei der Generalisierung habe ich auch die Möglichkeit der **Redefinition** von geerbten Merkmalen. Das bedeutet, dass Attribute, navigierbare Assoziationsenden, oder Operationen, die geerbt werden, spezifischer gemacht werden - ich redefiniere sie. Ich muss mich aber dabei in einem gewissen Rahmen bewegen, ich kann nicht einfach irgendetwas hinschreiben.

Beispielsweise: Wenn ich aus der Oberklasse ein Attribut von einem gewissen Typ erbe, so darf beim Redefinieren nicht irgendein anderer Typ angenommen werden, sondern es muss ein Subtyp des übergeordneten Typs sein. Das sehen Sie nachher am Beispiel.

Habe ich beim Assoziationsende eine Multiplizität von z.B. 0 bis * angegeben, dann kann ich das redefinieren auf 1 bis *, die redefinierte Multiplizität muss aber innerhalb des Intervalls der Oberklasse liegen.

Beim Überschreiben einer Operation wird gefordert, dass sie zumindest die gleiche Anzahl von Parametern aufweisen muss.

Generalisierung: Redefinition von geerbten Merkmalen (2/2)



Sehen wir uns die Redefinition von geerbten Merkmalen an einem Beispiel an.

In der Oberklasse „Eintrag“ gibt es ein Attribut „typ“ vom Typ „Eintragstyp“. Dieses Attribut wird in der Subklasse „Termin“ redefiniert. D.h. es gibt in der Subklasse auch einen Eintrag der „typ=Termin“ heißt und ich redefiniere jetzt hier den Typ.

Gleiches haben wir hier bei dem Attribut „prioritaet“, wobei auch der Typ des Parameters der Operation „setzePrioritaet“ entsprechend angepasst wurde.

Dieses Konzept der Redefinition wird allerdings nur in Spezialfällen benötigt und wird in der Praxis eher selten verwendet.

Exkurs: Ordnung und Eindeutigkeit von Assoziationen

- **Ordnung** {ordered} ist **unabhängig von Attributen**



- **Eindeutigkeit**

- Wie bei Attributen durch {unique} und {nonunique}
- Kombination mit Ordnung {set}, {bag}, {sequence} bzw. {seq}

Eindeutigkeit	Ordnung	Kombination	Beschreibung
<i>unique</i>	<i>unordered</i>	set	Menge (Standardwert)
<i>unique</i>	<i>ordered</i>	<i>orderedSet</i>	Geordnete Menge
<i>nonunique</i>	<i>unordered</i>	<i>bag</i>	Multimenge, d.h. Menge mit Duplikaten
<i>nonunique</i>	<i>ordered</i>	<i>sequence</i>	Geordnete Menge mit Duplikaten (Liste)



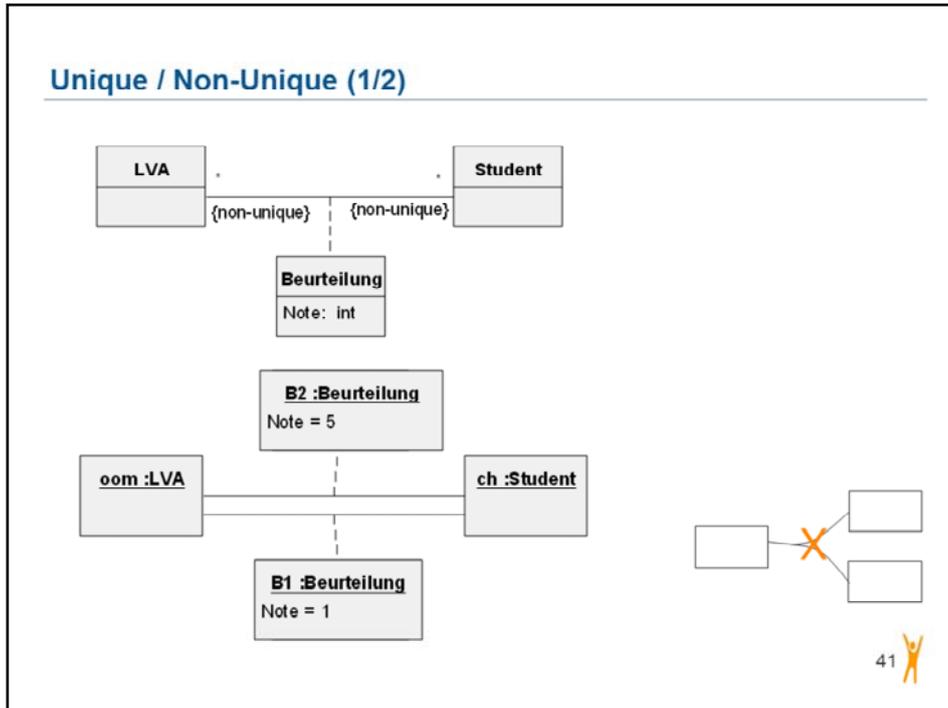
40

Ich möchte nun auf die **Ordnung und die Eindeutigkeit von Beziehungen** eingehen.

In dem dargestellten Beispiel haben wir eine Beziehung zwischen einer Klasse „Queue“ und einer Klasse „QueueItem“. Dabei kann es in einer „Queue“ 0 bis beliebig viele „QueueItems“ geben und umgekehrt befindet sich ein „QueueItem“ nur in einer „Queue“. Am Assoziationsende von „QueueItem“ wird normalerweise, wenn nichts anders angegeben ist, angenommen, dass die „QueueItems“ einer „Queue“ nicht geordnet sind. D.h., wenn eine Klasse eine Beziehung zu mehreren Objekten der anderen Klasse hat, ist es nur wichtig, dass diese Beziehungen existiert, aber es wird im Normalfall keine Rangfolge unter den Objekten angenommen, außer man gibt am Assoziationsende das Schlüsselwort „**ordered**“ an. Dann wird auch eine Reihenfolge vorgenommen, d.h. in unserem Fall sind die „QueueItems“ einer „Queue“ geordnet.

Ein weiterer wichtiger Punkt ist die Unterscheidung zwischen „**unique**“ und „**non-unique**“. Im Normalfall, wenn man nichts anderes angibt, ist ein Beziehungsende „unique“. Das bedeutet, ein „QueueItem“ darf in einer „Queue“ nur einmal vorkommen. Setze ich das Beziehungsende auf „non-unique“, dürfte das QueueItem öfters in der Queue vorkommen.

Unique / Non-Unique (1/2)



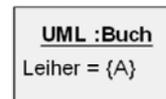
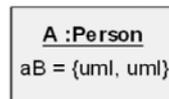
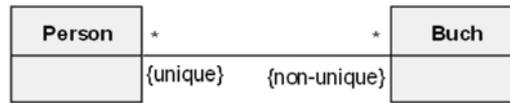
Um den Unterschied zwischen „unique“ und „non-unique“ zu verdeutlichen, schauen wir uns noch ein Beispiel an.

Wir haben die Klasse „Lehrveranstaltung“, die Klasse „Student“ und eine Beziehung zwischen ihnen, wobei eine „Lehrveranstaltung“ von mehreren „Studenten“ besucht wird und ein „Student“ mehrere „Lehrveranstaltungen“ besucht. Außerdem gibt es eine Assoziationsklasse „Beurteilung“ mit dem Attribut „Note“.

Im ersten Moment würde ich sagen, dass die Lösung mit einer n:m-Beziehung ganz gut aussieht. Jetzt ist aber die Frage, was man unter einer Lehrveranstaltung versteht. Wenn ich darunter nur „Objektorientierte Modellierung“, also OOM, verstehe und nicht „OOM im Wintersemester 2010“, dann haben wir hier Studienverschärfung. Warum haben wir Studienverschärfung? Wir sehen uns die Instanzen an. „OOM“ ist eine Instanz von „Lehrveranstaltung“ und „CH“ ist eine Instanz von „Student“. Der Student „CH“ besucht die Lehrveranstaltung „OOM“, d.h. wir nehmen einen Link auf und es wird festgestellt, dass der Student „CH“ hier eine negative Note bekommt. D.h. es gibt die Lehrveranstaltung „OOM“, die besucht der Student „CH“ und wird negativ beurteilt. Und was natürlich der Student „CH“ jetzt gerne machen würde ist, die Lehrveranstaltung noch einmal zu besuchen, um sie dann hoffentlich positiv abzuschließen. Aber im Normalfall, wenn an einem Beziehungsende nichts angegeben wird, gilt es als „unique“ und „unique“ bedeutet, dass jeder „Student“ nur einen Link zu einer bestimmten „Lehrveranstaltung“ haben darf. Jetzt gibt es aber zwei Noten vom Student „CH“ für die Lehrveranstaltung „OOM“: Den Fünfer aus dem ersten Besuch und einen Einser aus dem zweiten Besuch. D.h. wenn bereits ein Link zwischen dem Student „CH“ und der Lehrveranstaltung „OOM“ mit der Note „N5“ besteht, kann ein weiterer Link zwischen ihnen mit der Note „S1“ nicht mehr aufgenommen werden. Um dieses Problem zu lösen, kann man auf dem Assoziationsende bei „Student“ „non-unique“ angeben. Wichtig ist, dass ich beide Seiten auf „non-unique“ setzen muss. Das sieht man bei der graphischen Darstellung schön: Ein „Student“ kann zweimal in der „Lehrveranstaltung“ vorkommen und umgekehrt muss beim „Studenten“ auch die „Lehrveranstaltung“ mehrmals vorkommen können, damit das Modell richtig ist. D.h. ich muss beide Assoziationsende auf „non-unique“ setzen.

Es gibt Fälle, wo ich die zweite Seite nicht auf „unique“ setzen muss, nur kann man es dann nicht mehr so schön graphisch im Objektdiagramm darstellen, weil es die Notationsform die unten rechts dargestellt ist, wo sich ein Link aufspaltet in zwei Links, leider nicht gibt. Aber programmiertechnisch könnte ich es natürlich schaffen, dass ich auf der einen Seite Duplikate in der Menge erlaube und auf der anderen Seite Duplikate herausnehmen muss.

Unique / Non-Unique (2/2)



Ich möchte jetzt noch ein Beispiel zu „unique“ und „non-unique“ bringen.

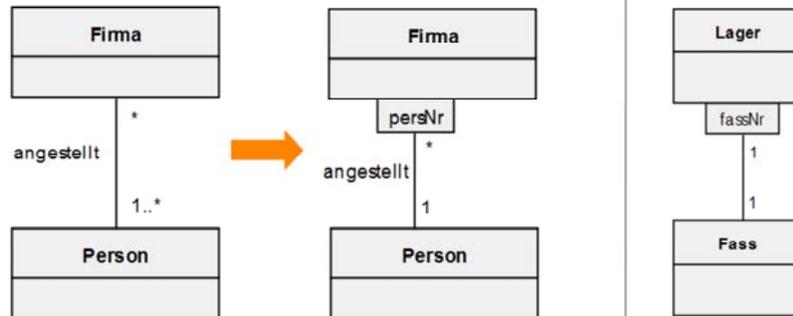
Es gibt die Klassen „Person“ und „Buch“. Eine Person kann mehrere Bücher ausborgen. Ein Buch kann zwar nicht gleichzeitig aber zumindest über einen Zeitraum auch von mehreren Personen ausgeborgt werden. Dabei könnte man auf der Seite von „Buch“ „non-unique“ und auf der Seite der „Person“ „unique“ definieren. Jetzt ist die Frage, ob das auch gültig ist. Angenommen die Person „A“ borgt sich das Buch „UML@work“ aus. Dann besitzt die Person „A“ ein Attribut „ausgeborgteBuecher“, das eine Menge von Referenzen zu allen ausgeborgten Büchern besitzt. In dieser Menge ist dann auch der Link zu dem Buch „UML@work“ enthalten. Und ebenso muss das Buch „UML@work“ ein Attribut „Leiherr“ besitzen, das eine Menge von Links auf die Ausborger oder Leiherr des Buches beinhaltet und hier ist auch die Person „A“ eingetragen.

Gibt „A“ jetzt das Buch zurück und borgt es sich noch einmal aus, so wird zwar das Buch noch einmal in die Liste der ausgeborgten Bücher der Person „A“ eingetragen, jedoch wird „A“ nicht erneut in die Liste der Leiherr des Buchs „UML@work“ eingetragen weil wir hier „unique“ angegeben haben.

Programmtechnisch ist es kein Problem so etwas abzubilden. „unique“ und „non-unique“ sind also nicht Merkmale der Beziehung, sondern jeweils des Beziehungsendes. Aber graphisch kann man das nicht mehr mit Hilfe von Links in einem Objektdiagramm darstellen.

Exkurs: Qualifizierte Assoziation (1/2)

- Besteht aus einem **Attribut** oder einer **Liste von Attributen**, deren Werte die Objekte der assoziierten Klasse partitionieren
- **Reduziert** meist die **Multiplizität**
- Stellt eine **Eigenschaft der Assoziation** dar



43 

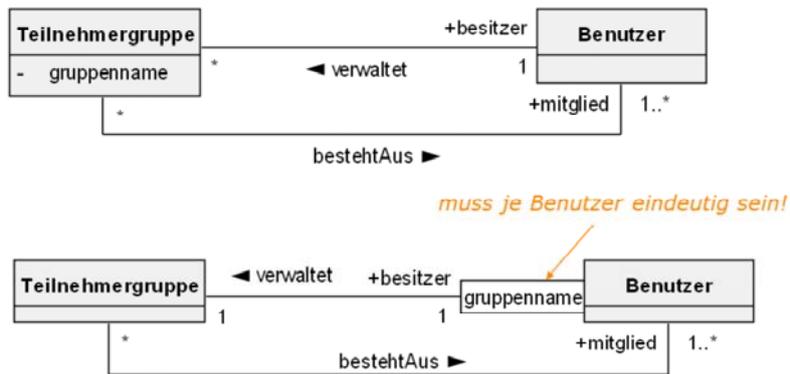
Als nächstes folgt das Konzept der **qualifizierten Assoziation**. Dieses wollen wir an einem Beispiel beschreiben.

In einer „Firma“ arbeiten ein bis mehrere „Personen“. Ich möchte aber zusätzlich noch eine „Personalnummer“ aufnehmen, wobei jeder Mitarbeiter eine eindeutige Personalnummer besitzt. Würde ich im Klassendiagramm, das auf der linken Seite dargestellt ist, die Personalnummer als Attribut aufnehmen, so könnte man allerdings nicht ausschließen, dass in einer Firma mehrere Personen mit der selben Personalnummer arbeiten.

Um darzustellen, dass die Personalnummer eindeutig ist, kann man auf der Seite der Firma ein qualifizierendes Attribut „Personalnummer“ aufnehmen. Dann arbeiten in einer Firma mit einer Personalnummer nicht mehrere Personen, sondern es arbeitet nur eine Person mit dieser Personalnummer in dieser Firma. Daher ist die Multiplizität auf der Seite „Person“ nun 1. D.h. ein qualifizierendes Attribut reduziert meist die Multiplizität.

Ein anderes Beispiel ist das folgende: Es gibt die Klasse „Lager“ und es gibt die Klasse „Fass“ und ich möchte jetzt darstellen, dass es zwar mehrere Fässer mit der Nummer 5 gibt, aber es innerhalb eines Lagers nur ein Fass mit der Nummer 5 gibt. Das stellt man so dar, dass die „Fassnummer“ als qualifizierendes Attribut von „Lager“ aufgenommen wird und in einem Lager gibt es dann mit einer Fassnummer nur ein Fass und ein Fass befindet sich nur in einem Lager.

Exkurs: Qualifizierte Assoziation (2/2)



Hier ist noch ein Beispiel.

Wir haben hier einen „Benutzer“ und eine „Teilnehmergruppe“. Ein Benutzer verwaltet mehrere Teilnehmergruppen. Eine Teilnehmergruppe wird von einem Benutzer verwaltet. Wenn der Gruppenname je Benutzer eindeutig sein muss, so ist „Gruppenname“ ein qualifizierendes Attribut und ein Benutzer verwaltet mit einem Gruppennamen jetzt nur mehr eine Teilnehmergruppe. Die untere Beziehung ist davon nicht betroffen: In einer „Teilnehmergruppe“ befinden sich mehrere „Benutzer“ und ein „Benutzer“ kann in mehreren „Teilnehmergruppen“ teilnehmen. Bitte beachten Sie: Der Benutzer verwaltet in dem unteren Klassendiagramm auch mehrere Teilnehmergruppen. Nur mit einem Namen verwaltet er nur eine Teilnehmergruppe.

Dieses Konzept wird nur eingesetzt, wenn man davon ausgehen kann, dass die Multiplizität ohne qualifizierendem Attribut ein Stern * wäre, weil sonst das ganze Konzept keinen Sinn macht. Geht man davon aus, dass ein Benutzer ohnehin nur eine Teilnehmergruppe verwaltet, dann bräuchte man kein qualifizierendes Attribut. Dieses würde keinen Sinn ergeben. Sinnvoll wird es nur eingesetzt, wenn das qualifizierende Attribut dazu führt, dass sich eine Reduktion der Multiplizitäten ergibt. Meistens eine Reduktion auf 1.

Aufgabenstellung

- Gesucht ist ein vereinfacht dargestelltes Modell der TU Wien entsprechend der folgenden Spezifikation.

Die TU besteht aus mehreren Fakultäten, die sich wiederum aus verschiedenen Instituten zusammensetzen. Jede Fakultät und jedes Institut besitzt eine Bezeichnung. Für jedes Institut ist eine Adresse bekannt. Jede Fakultät wird von ihrem Dekan, einem Mitarbeiter, geleitet.

Die Gesamtanzahl der Mitarbeiter ist bekannt. Mitarbeiter haben eine Sozialversicherungsnummer, einen Namen und eine E-Mail-Adresse. Es wird zwischen wissenschaftlichem und nicht-wissenschaftlichem Personal unterschieden.

Wissenschaftliche Mitarbeiter sind zumindest einem Institut zugeordnet.

Für jeden wissenschaftlichen Mitarbeiter ist seine Fachrichtung bekannt.

Weiters können wissenschaftliche Mitarbeiter für eine gewisse Anzahl an Stunden an Projekten beteiligt sein, von welchen ein Name und Anfangs- und Enddatum bekannt sind.

Manche wissenschaftliche Mitarbeiter führen Lehrveranstaltungen durch – diese werden als Vortragende bezeichnet. LVAs haben eine ID, einen Namen und eine Stundenanzahl.



Nun kommen wir zu einem umfassenderen Beispiel. Hier sehen Sie die Aufgabenstellung.

Identifikation von Klassen (1/2)

Die TU besteht aus mehreren **Fakultäten**, die sich wiederum aus verschiedenen **Instituten** zusammensetzen. Jede Fakultät und jedes Institut besitzt eine Bezeichnung. Für jedes Institut ist eine Adresse bekannt. Jede Fakultät wird von ihrem Dekan, einem Mitarbeiter, geleitet. Die Gesamtanzahl der **Mitarbeiter** ist bekannt. Mitarbeiter haben eine Sozialversicherungsnummer, einen Namen und eine E-Mail-Adresse. Es wird zwischen **wissenschaftlichem** und **nicht-wissenschaftlichem Personal** unterschieden.

Wissenschaftliche Mitarbeiter sind zumindest einem Institut zugeordnet.

Für jeden wissenschaftlichen Mitarbeiter ist seine Fachrichtung bekannt.

Weiters können wissenschaftliche Mitarbeiter für eine gewisse Anzahl an Stunden an **Projekten** beteiligt sein, von welchen ein Name und Anfangs- und Enddatum bekannt sind.

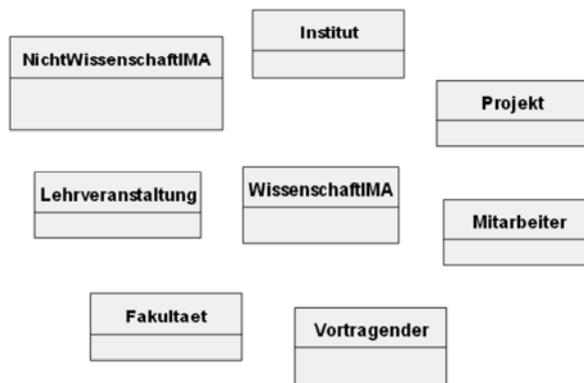
Manche wissenschaftliche Mitarbeiter führen **Lehrveranstaltungen** durch – diese werden als **Vortragende** bezeichnet. LVAs haben eine ID, einen Namen und eine Stundenanzahl.



Auf dieser Folie sind nun im Text die Klassennamen fett hervorgehoben.

Die TU besteht aus mehreren **Fakultäten**, die sich wiederum aus verschiedenen **Instituten** zusammensetzen. Ein Institut gehört also automatisch genau zu einer Fakultät und kann auch alleine gar nicht bestehen. Die TU wird selbst nicht modelliert, da das Modell ja für die TU erstellt wird. Jede Fakultät und jedes Institut besitzt eine Bezeichnung. Für jedes Institut ist eine Adresse bekannt. Jede Fakultät wird von Ihrem **Dekan**, einem **Mitarbeiter**, einem besonderen Mitarbeiter, geleitet. Die Gesamtzahl der Mitarbeiter ist bekannt. Mitarbeiter haben eine Sozialversicherungsnummer, einen Namen und eine E-Mail-Adresse. Wir unterscheiden zwischen **wissenschaftlichem** und **nicht-wissenschaftlichem Personal**. Wissenschaftliche Mitarbeiter sind zumindest einem Institut zugeordnet, können aber auch mehreren Instituten zugeordnet werden. Für jeden wissenschaftlichen Mitarbeiter ist seine Fachrichtung bekannt. Weiters können wissenschaftliche Mitarbeiter für eine gewisse Anzahl an Stunden an **Projekten** beteiligt sein, von welchen ein Name und Anfangs- und Enddatum bekannt sind. Manche wissenschaftlichen Mitarbeiter führen **Lehrveranstaltungen** durch, diese werden dann als **Vortragende** bezeichnet, und Lehrveranstaltungen haben eine eindeutige ID, einen Namen und eine Stundenzahl.

Identifikation von Klassen (2/2)



Aus den fetten hervorgehobenen Wörtern ergeben sich also die Klassen, d.h. wir haben „Fakultaet“ und „Institut“, wir haben „Mitarbeiter“, wobei wir unterscheiden zwischen wissenschaftlichen Mitarbeitern „WissenschaftlMA“ und nicht-wissenschaftlichen Mitarbeitern „NichtWissenschaftlMA“, weiters haben wir „Projekte“ und wir haben „Lehrveranstaltungen“, die von „Vortragenden“ gehalten werden.

Identifikation von Attributen (1/2)

Die TU besteht aus mehreren Fakultäten, die sich wiederum aus verschiedenen Instituten zusammensetzen. Jede Fakultät und jedes Institut besitzt eine **Bezeichnung**. Für jedes Institut ist eine **Adresse** bekannt. Jede Fakultät wird von ihrem Dekan, einem Mitarbeiter, geleitet.

Die **Gesamtanzahl** der Mitarbeiter ist bekannt. Mitarbeiter haben eine **Sozialversicherungsnummer**, einen **Namen** und eine **E-Mail-Adresse**. Es wird zwischen wissenschaftlichem und nicht-wissenschaftlichem Personal unterschieden.

Wissenschaftliche Mitarbeiter sind zumindest einem Institut zugeordnet.

Für jeden wissenschaftlichen Mitarbeiter ist seine **Fachrichtung** bekannt.

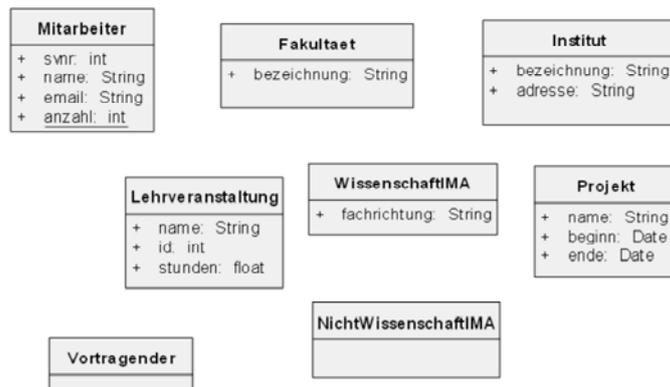
Weiters können wissenschaftliche Mitarbeiter für eine **gewisse Anzahl an Stunden** an Projekten beteiligt sein, von welchen ein **Name** und **Anfangs- und Enddatum** bekannt sind.

Manche wissenschaftliche Mitarbeiter führen Lehrveranstaltungen durch - diese werden als Vortragende bezeichnet. LVAs haben eine **ID**, einen **Namen** und eine **Stundenanzahl**.



Aus dem Text können auch relativ einfach die Attribute identifiziert werden, die zu jeder dieser Klassen gehören. Auf dieser Folie sind im Text die Attribute fett hervorgehoben.

Identifikation von Attributen (2/2)



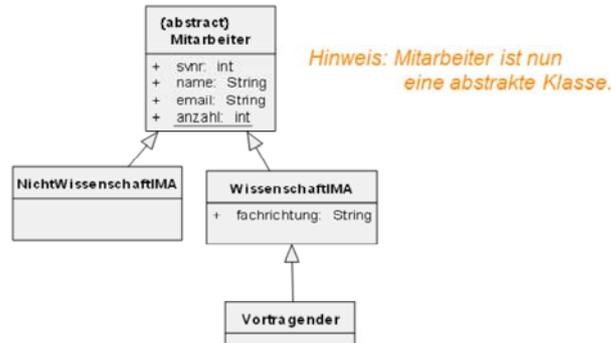
Die „Fakultät“ hat eine „Bezeichnung“. Das „Institut“ hat eine „Bezeichnung“ und eine „Adresse“ usw. Ich glaube ich muss das nicht für alle Klassen durchmachen.

Nur ein Punkt der vielleicht entscheidend ist: In der Angabe steht „Die Gesamtanzahl der Mitarbeiter ist bekannt“. Somit ist das Attribut „anzahl“ der Klasse „Mitarbeiter“ ein **Klassenattribut**, das entspricht einem „static“ Attribut in der Programmierung, da der Wert dieses Attributs nicht für jeden Mitarbeiter verschieden ist, sondern für alle Instanzen von Mitarbeiter gleich ist. Um auszudrücken, dass es sich um ein Klassenattribut handelt, wird es unterstrichen.

Sie fragen sich jetzt vielleicht, warum alle Attribute in diesem Beispiel als „public“ deklariert sind. Wenn man ein reines Datenmodell entwirft, dann setzt man die Attribute meistens auf „public“. Wenn man jedoch ein Modell für die Programmierumgebung entwirft, wird man sie eher als „private“ definieren, da auf sie wahrscheinlich mit Operationen zugegriffen wird.

Generalisierung

- Es wird zwischen wissenschaftlichem und nicht-wissenschaftlichem Personal unterschieden.
- Manche wissenschaftliche Mitarbeiter führen Lehrveranstaltungen durch – diese werden als Vortragende bezeichnet.



50

Wir unterscheiden zwischen wissenschaftlichem und nicht-wissenschaftlichem Personal. Dementsprechend verwenden wir hier das Konzept der Vererbung. Bei wissenschaftlichen Mitarbeitern kommt ein Attribut „fachrichtung“ dazu. Weiters gibt es noch eine spezielle Form von Mitarbeitern, nämlich die „Vortragenden“, die Lehrveranstaltungen halten und diese sind Spezialformen von wissenschaftlichen Mitarbeitern.

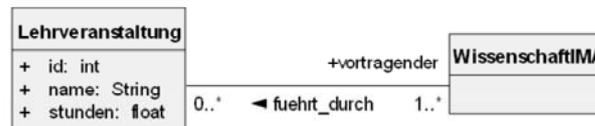
Bitte beachten Sie, dass wir mit diesem Beispiel die Konzepte erläutern wollen. Deshalb haben wir es mit den Vererbungshierarchien vielleicht immer ein bisschen überzogen. In der Praxis werden Sie eher wenige Ebenen der Generalisierung einsetzen, weil das in der Programmierung sonst nicht vernünftig handhabbar ist. Eines steht für mich immer fest: Es muss zumindest immer ein Attribut, eine Operation oder eine Assoziation in der Subklasse dazukommen, da sonst die Vererbung keinen Sinn macht. D.h. in dem Beispiel könnte ich den nicht-wissenschaftlichen Mitarbeiter vielleicht auch weg lassen. Dann dürfte ich natürlich die Klasse „Mitarbeiter“ nicht abstrakt machen. Man hätte dann eben eine unvollständige Aufteilung.

Assoziation (1/2)

- Manche wissenschaftliche Mitarbeiter führen Lehrveranstaltungen durch – diese werden als Vortragende bezeichnet.



- Alternative Modellierung:



51 

Hier ist die Assoziation zwischen Lehrveranstaltungen und den wissenschaftlichen Mitarbeitern, die auch als Vortragende fungieren, in zwei Varianten dargestellt.

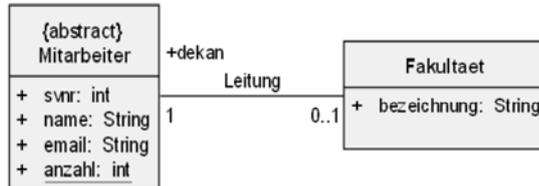
In der ersten Variante gibt es „Vortragende“ als eigene Klasse, die eine Sonderform von wissenschaftlichen Mitarbeitern sind und diese halten Lehrveranstaltungen. Ein Vortragender hält mindestens eine Lehrveranstaltung, weil sonst wäre er ja kein Vortragender. Aber unter Umständen hält er auch mehrere Lehrveranstaltungen. Eine Lehrveranstaltung wird mindestens von einem, aber eventuell auch von mehreren „Vortragenden“ gehalten.

Man könnte sich aber auch die Subklassenbildung ersparen und sagen, es gibt „wissenschaftliche Mitarbeiter“ und es gibt „Lehrveranstaltungen“ und wissenschaftliche Mitarbeiter halten nicht unbedingt eine Lehrveranstaltung. Es gibt welche, die halten keine, oder natürlich können sie auch, wie oben schon gesagt, mehrere Lehrveranstaltungen halten, daher 0 bis *. Umgekehrt wird eine Lehrveranstaltung noch immer von ein bis mehreren wissenschaftlichen Mitarbeitern gehalten. Ein wissenschaftlicher Mitarbeiter, der eine Lehrveranstaltung hält, d.h. hier an der Beziehung teilnimmt, nimmt die Rolle des „Vortragenden“ ein. D.h., bei der Beziehung Lehrveranstaltung – wissenschaftlicher Mitarbeiter, spielt der wissenschaftliche Mitarbeiter die Rolle „Vortragender“, welche am Beziehungsende notiert wird.

Beide Möglichkeiten wie im oberen und unteren Beispiel dargestellt sind valide.

Assoziation (2/2)

- Jede Fakultät wird von ihrem Dekan, einem Mitarbeiter, geleitet.



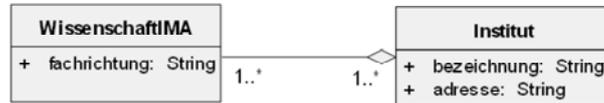
Jede Fakultät wird von ihrem Dekan, einem speziellen Mitarbeiter, geleitet.

Man könnte hier natürlich wieder eine Subklasse „Dekan“ von „Mitarbeiter“ einführen, auch wenn es etwas zweifelhaft ist für die wenigen Dekane eine eigene Subklasse zu bilden. Aber wenn man das möchte, könnte man es machen.

Hier ist die zweite Variante dargestellt. Ein „Mitarbeiter“ leitet 0 bis eine „Fakultät“, eine „Fakultät“ wird von genau einem „Mitarbeiter“ geleitet und dieser Mitarbeiter hat dann die Rolle des „Dekans“.

Schwache Aggregation

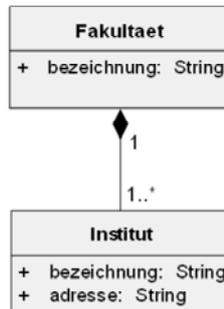
- Wissenschaftliche Mitarbeiter sind zumindest einem Institut zugeordnet.



Wissenschaftliche Mitarbeiter sind zumindest einem Institut zugeordnet – zumindest heißt, es könnten auch mehrere Institute sein. Ein Institut besteht aus einem bis mehreren Mitarbeitern. Ein wissenschaftlicher Mitarbeiter kann aber auch an mehreren Instituten arbeiten, daher auch auf dieser Seite 1 bis *. Es ist klar, dass das keine Komposition ist, weil der Mitarbeiter eben auch an mehreren Instituten arbeiten kann. Daher kann es, wenn überhaupt, nur eine schwache Aggregation sein und die kann man eventuell wegdiskutieren. Ich verwende vor allem deshalb eine Aggregation, weil wir an diesem Beispiel später noch die Code-Transformation erklären wollen.

Starke Aggregation

- Die TU besteht aus mehreren Fakultäten, die sich wiederum aus verschiedenen Instituten zusammensetzen.



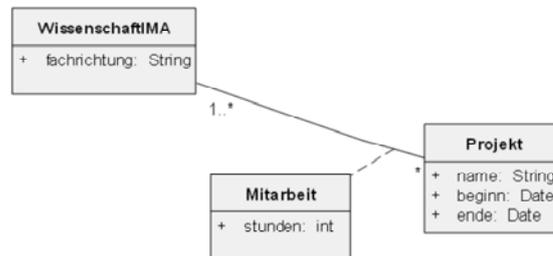
Hier hingegen macht die starke Aggregation, Komposition genannt, Sinn: Die TU besteht aus mehreren Fakultäten, die sich wiederum aus verschiedenen Instituten zusammensetzen.

Eine Fakultät besteht also aus mindestens einem Institut, aber sehr oft aus mehreren. Ein Institut ist genau einer Fakultät zugeordnet und wenn eine Fakultät gelöscht wird, werden wir auch die Institute dieser Fakultät löschen, denn bei der Komposition wird das Löschen ja propagiert.

Ein Institut kann in unserem Fall auch nicht alleine bestehen, es gibt keine freischwimmenden Institute in unserer Universität, sondern sie können nur innerhalb der Fakultät existieren. Daher ist die Multiplizität auch nicht 0 bis 1 sondern wirklich 1 auf Seiten der Raute. Hier haben wir also wirklich eine Existenz-Abhängigkeit.

Assoziationsklasse

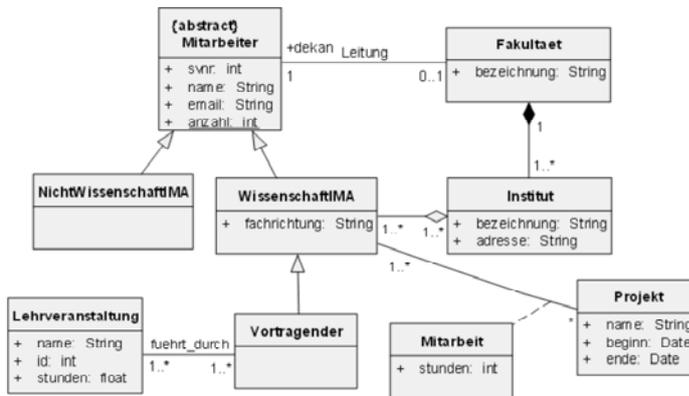
- Weiters können wissenschaftliche Mitarbeiter an Projekten beteiligt sein.



Im Beispiel gibt es auch noch eine Assoziationsklasse. Für die Beziehung wo wissenschaftliche Mitarbeiter an Projekten beteiligt sind.

Ein Mitarbeiter arbeitet an 0 bis mehreren Projekten und ein Projekt hat mindestens einen Mitarbeiter oder auch mehrere. Die Mitarbeit in Stunden wird hier als Attribut gefordert. Dieses Attribut kann man nicht bei Mitarbeiter dazugeben, weil sonst ein Mitarbeiter in jedem Projekt gleich viele arbeiten würde. Man kann die Stunden auch nicht beim Projekt dazugeben, weil sonst alle Mitarbeiter des Projekts gleich viel arbeiten würden. Und daher sind die Stunden ein Merkmal der Beziehung und werden über die Assoziationsklasse „Mitarbeit“ an die Beziehung hinzugefügt.

... das gesamte Diagramm



Zum Abschluss des Beispiels sehen wir hier das gesamte Klassendiagramm noch auf einer Folie dargestellt.

Übersetzung nach Java: Klassen (1/2)



```
class Lehrveranstaltung {
    public String name;
    public int id;
    public float stunden;

    public Lehrveranstaltung(String name, int id,
                             float stunden) {
        this.name = name;
        this.id = id;
        this.stunden = stunden;
    }
}
```

- Erstellung einer konkreten Instanz oom:

```
Lehrveranstaltung oom = new Lehrveranstaltung(
    "Objektorientierte Modellierung", 394, 4);
```

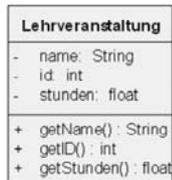
- Zugriff auf Attribut name: oom.name;

Nun kommen wir zur **Übersetzung nach Java**. Der Code hier ist nur beispielhaft um das Verständnis zu unterstützen. UML selbst liefert nur eine graphische Syntax für die Modellierung. Eine Übersetzung in eine Programmiersprache ist nicht Teil der UML und ist auch nicht standardisiert. Aber üblicherweise bietet Ihnen jedes UML-Tool eine Übersetzung in eine Reihe von Programmiersprachen an und man muss sich dann die konkrete Übersetzung anschauen und bewerten, ob man damit einverstanden ist.

Auf dieser Folie sehen wir, wie Klassen nach Java übersetzt werden können. Eine Klasse „Lehrveranstaltung“ in UML wird natürlich im Code auch zu einer Klasse „Lehrveranstaltung“. Auch die Attribute werden in die Klasse übernommen, wobei der Sichtbarkeitsoperator berücksichtigt wird, also das Plus + wird zu „public“. Auch die Datentypen, die wir am Ende der Einheit noch durchführen werden, und die Namen der Attribute werden übernommen.

Im Code ist auch noch ein Konstruktor angegeben. Ob es sinnvoll ist, im Klassendiagramm Konstruktoren anzugeben, oder nicht, liegt vor allem an der Transformations-Engine in Ihrem Tool. Denn werden die Konstruktoren automatisch von der Engine erzeugt, wird man sie nicht im Diagramm modellieren, vor allem nicht den Default-Konstruktor. Soll eine Klasse mehrere Konstruktoren haben, dann macht es wahrscheinlich Sinn, auch die nicht Default-Konstruktor im Diagramm anzugeben. In unserem Beispiel gehen wir davon aus, dass der Konstruktor nicht angegeben werden muss, weil im Diagramm haben wir keinen angegeben und trotzdem ist im Code sozusagen ein Default-Konstruktor erzeugt worden.

Übersetzung nach Java: Klassen (2/2)



```
class Lehrveranstaltung {  
  
    private String name;  
    private int id;  
    private float stunden;  
  
    public Lehrveranstaltung(String name, int id,  
                             float stunden) {  
  
        this.name = name;  
        this.id = id;  
        this.stunden = stunden;  
    }  
    public String getName() { return name; }  
    public int getId() { return id; }  
    public float getStunden() { return stunden; }  
}
```

- *Erstellung einer Instanz oom: wie vorher*
- *Zugriff auf Attribut name: `oom.name`; `oom.getName()`;*



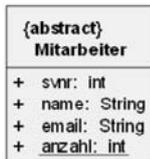
© BIG / TU Wien



Hier haben wir mehr oder weniger wieder das gleiche Beispiel, nur diesmal haben wir alle Attribute in UML auf „private“ gesetzt, dementsprechend sind auch die Attribute in Java „private“. „public“ sind dann nur die Operationen, mit Hilfe derer ich auf die Attribute zugreife.

Ob es Sinn macht, getter- und setter-Funktionen in UML zu definieren, hängt wiederum von der verwendeten Transformations-Engine ab. Wenn Sie irgendwo eine Einstellung in ihrem Tool vornehmen können, dass getter und setter automatisch erzeugt werden sollen, dann werden Sie die getter und setter nicht immer beim Modell aufnehmen. Die Tools bieten Ihnen auch üblicherweise, selbst wenn es nicht im UML Standard vorgesehen ist, die Möglichkeit, dass Sie für jedes einzelne Attribut definieren können, ob getter und setter erzeugt werden sollen. Das erspart schon relativ viel Schreibarbeit und das ist ja auch eine der Aufgaben der Modellierung.

Übersetzung nach Java: Abstrakte Klasse



```
abstract class Mitarbeiter {
    public int svnr;
    public String name;
    public String email;
    public static int anzahl = 0;

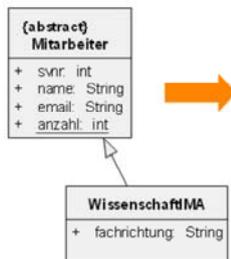
    public Mitarbeiter(int svnr, String name,
                       String email) {
        this.svnr = svnr;
        this.name = name;
        this.email = email;
    }
}
```

- Nicht möglich: ~~mal=new Mitarbeiter(123,"abc", "abc@xyz.at");~~



Die Übersetzung der abstrakte Klassen „Mitarbeiter“, die wir hier haben, ist auch keine Überraschung. Eine abstrakte Klasse in UML wird auch zu einer abstrakten Klasse in Java. Der Konstruktor der abstrakten Klasse kann natürlich selbst nicht aufgerufen werden, das wird hoffentlich auch keine Überraschung für Sie sein.

Übersetzung nach Java: Generalisierung



```
class WissenschaftlMA extends Mitarbeiter {
    public String fachrichtung;

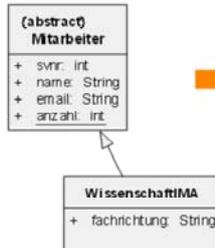
    public WissenschaftlMA(int svnr, String name,
        String email, String fachrichtung) {
        super(svnr, name, email);
        this.fachrichtung = fachrichtung;
    }
}
```

- Neue Instanz: `wma1 = new WissenschaftlMA(123, "abc", "abc@xyz.at", "Informatik");`
- Zugriff auf Name: `wma1.name;`

Hier ist die Übersetzung der Generalisierung dargestellt. Sie enthält auch keine großen Überraschungen. Wir sehen hier den Code der Subklasse „wissenschaftlicher Mitarbeiter“, welche von der Klasse „Mitarbeiter“, die wir auf der vorigen Folie gesehen haben, erbt. Aus der Vererbungsbeziehung wird der Code „class <<Name der Subklasse>> extends <<Name der Superklasse>>“ erzeugt. Dann haben wir da noch den Konstruktor angegeben, in dem der Konstruktor der Superklasse aufgerufen wird. Dementsprechend kann man natürlich jetzt einen neuen wissenschaftlichen Mitarbeiter anlegen und z.B. auch auf seinen Namen zugreifen, weil der Namen als „public“ definiert ist, und damit von der Superklasse „Mitarbeiter“ geerbt wird.

Übersetzung nach Java: Klassenvariable

- Angabe: Die Anzahl der Mitarbeiter ist bekannt
- Realisierung: Klassenvariable



```
abstract class Mitarbeiter {
    ...
    public static int anzahl = 0;

    public Mitarbeiter (...) {
        ...
        anzahl++;
    }
}
class WissenschaftlMA extends Mitarbeiter {
    ...
    public WissenschaftlMA (...) {
        super(...);
    }
}
```

- Wird eine Klasse, die von Mitarbeiter erbt, instanziiert, wird `anzahl` um 1 erhöht
- Durch `Mitarbeiter.anzahl` oder `WissenschaftlMA.anzahl` oder `wmal.anzahl` bekommt man die Anzahl der Mitarbeiter



Hier haben wir jetzt die Klasse „Mitarbeiter“ nochmal im Detail dargestellt und zwar betrachten wir hier die Klassenvariable „anzahl“ vom Typ Integer „int“. Die Klassenvariable wird übersetzt in ein „static“ Attribut und dementsprechend haben wir auch hier den Code „public static int anzahl“, und „anzahl“ wird mit 0 initialisiert. Bei jedem Anlegen eines Mitarbeiters, d.h. bei jedem Konstruktor-Aufruf von „Mitarbeiter“, wird das Attribut „anzahl“ um 1 erhöht.

Wissenschaftlicher Mitarbeiter ist natürlich auch ein Mitarbeiter und dementsprechend wird beim Anlegen von einem wissenschaftlichen Mitarbeiter ebenfalls das Attribut „anzahl“ um 1 erhöht, indem der Konstruktor der Superklasse „Mitarbeiter“ im Konstruktor der Subklasse „wissenschaftlicher Mitarbeiter“ aufgerufen wird.

Übersetzung nach Java: 1:1-Assoziation



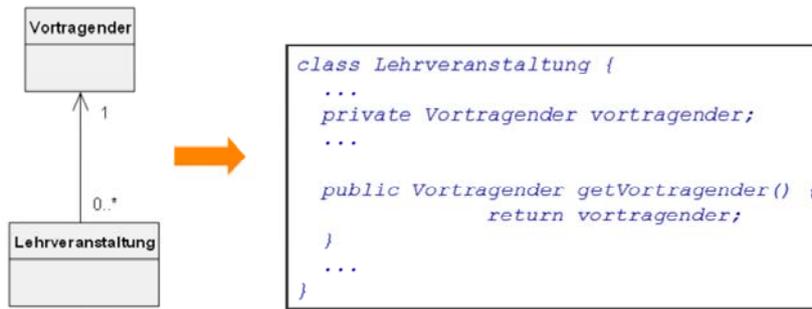
```
class Dekan {
    ...
    private Fakultaet rFakultaet;
    ...
    public Fakultaet getFakultaet() {
        return rFakultaet; }
}
class Fakultaet {
    ...
    private Dekan rDekan;
    ...
    public Dekan getDekan() { return rDekan; }
}
```

- Die entsprechenden Rollen werden als Attribute in die jeweils gegenüberliegende Klasse eingefügt
- Ist die Multiplizität 0..1, kann das entsprechende Attribut unter Umständen auch den Wert `null` haben

Jetzt kommen wir zu etwas, das ein bisschen spektakulärer ist. Wir haben hier eine Beziehung von „Dekan“ zur „Fakultät“. Ein Dekan leitet eine Fakultät und eine Fakultät wird von einem Dekan geleitet. Bei dieser Beziehung sind auch Rollen angegeben. Für jede Rolle kann auch ein Sichtbarkeitsoperator angegeben werden: „private“, „public“ oder „protected“. Der Sichtbarkeitsoperator macht aber auch ohne Rollenangabe Sinn. In unserem Beispiel gehen wir davon aus, dass es nach Best Practice modelliert ist, d.h. ich kann in beide Richtungen navigieren und dementsprechend habe ich von „Fakultät“ nach „Dekan“ eine Referenz zu setzen, weil von „Fakultät“ soll ich zum „Dekan“ navigieren können. Was bedeutet das? Ich muss bei der Fakultät ein Attribut aufnehmen, das auf den Dekan zeigt. Wie nenne ich das Attribut und wie ist die Sichtbarkeit von diesem Attribut? Das wird durch die Rollenangabe festgelegt: Es gibt also die Klasse „Fakultät“ und ich nehme das Attribut „rDekan“ auf. Es ist vom Typ „Dekan“ und ist „private“, weil bei der Rolle ein Minus - angegeben ist. Umgekehrt auf der anderen Seite möchte man auch vom „Dekan“ zur „Fakultät“ navigieren können. Wir erzeugen daher in der Klasse „Dekan“ ein Attribut mit dem Namen „rFakultaet“ (kurz für „Referenz Fakultät“). „rFakultaet“ entspricht der Rollenangabe auf der Seite des Dekans. Bei dieser Rolle ist der Sichtbarkeitsoperator ein Minus -, daher ist die Sichtbarkeit des Attributs im Code ebenfalls „private“.

Sind also bei einer 1:1 Beziehung beide Enden navigierbar, so erzeugen wir im Code je eine Instanzvariable, die uns eine Referenz auf die andere Seite ermöglicht.

Übersetzung nach Java: Unidirektionale ?:1-Assoziation



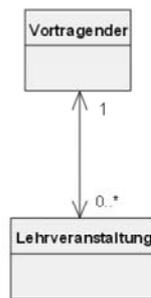
- Keine Änderung an Vortragender!
- Viel leichter zu implementieren als bidirektionale Assoziation

Hier haben wir die Übersetzung nach Java bei einer unidirektionalen n:1 Assoziation. Eine „Lehrveranstaltung“ wird von einem „Vortragenden“ gehalten. In diese Richtung wollen wir navigieren. Umgekehrt, ein „Vortragender“ kann zwar mehrere „Lehrveranstaltungen“ halten, aber diese Information ist nutzlos, weil wir vom „Vortragenden“ nicht zur „Lehrveranstaltung“ navigieren wollen.

Wir müssen daher in der Klasse „Lehrveranstaltung“ eine Referenz zum Vortragenden aufnehmen. Nun ist aber auf Seite des Vortragenden im Klassendiagramm kein Rollename angegeben. Es liegt dann an der Transformations-Engine, einen Attributnamen zu vergeben. Wir nennen ihn „vortragender“, aber klein geschrieben in unserem Fall. Auch ist kein Sichtbarkeitsoperator auf Seiten des Vortragenden angegeben und dementsprechend liegt es wieder an der Transformations-Engine, festzulegen, ob ein Attribut „private“, „public“ oder „protected“ ist. Wir nehmen für unserem Fall an, dass wenn im UML-Diagramm kein Sichtbarkeitsoperator angegeben ist, das entstehende Attribut dann „private“ ist. Umgekehrt ist es so, dass ich von der Klasse „Vortragender“ nicht zur Klasse „Lehrveranstaltung“ navigieren kann. Dementsprechend wird in der Klasse „Vortragender“ kein zusätzliches Attribut aufgenommen, denn eine Referenz zur Lehrveranstaltung ist nicht nötig.

Auf dieser Folie steht am Ende „Viel leichter zu implementieren als bidirektionale Assoziation“, welche wir auf der nächsten Folie vorstellen werden. Diese Behauptung ist mit Vorsicht zu genießen. Natürlich sollte jeder von Ihnen in der Lage sein, eine bidirektionale Assoziation zu implementieren. Außerdem bestimmt ja nicht der Programmierer ob eine Assoziation in eine oder in beide Richtungen navigierbar sein soll, sondern das ist in den Requirements festgelegt. Ich sage das nur, damit niemand in der Übung auf die Idee kommt, ein navigierbares Ende wegzulassen, weil es unter Anführungszeichen „leichter“ ist.

Übersetzung nach Java: Bidirektionale 1:*-Assoziation



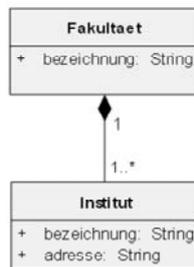
```
java.util.ArrayList;
class Vortragender {
    ...
    private ArrayList lvas;
    ...

    public Vortragender {
        lvas = new ArrayList();
    }
    ...
}
```

- Lehrveranstaltung wird wie vorher implementiert
- Wenn n fix vorgegeben ist (nicht *), dann kann auch ein Array verwendet werden

Wenn man in beide Richtungen navigieren kann, dann benötigt man natürlich eine Referenz von „Vortragender“ zur „Lehrveranstaltung“. Da ein Vortragender mehrere Lehrveranstaltungen abhalten kann, muss ich hier auch einen Typ wählen, der es mir ermöglicht, mehrere Objekte der anderen Seite aufzunehmen. Ein Beispiel wäre hier eine „ArrayList“. Dementsprechend ist das entstehende Attribut, das wir „lvas“ nennen, nicht vom Typ „Lehrveranstaltung“, weil das könnte nur eine Einzel-Referenz aufnehmen, sondern vom Typ „ArrayList“ mit der Sichtbarkeit „private“.

Übersetzung nach Java: Starke Aggregation

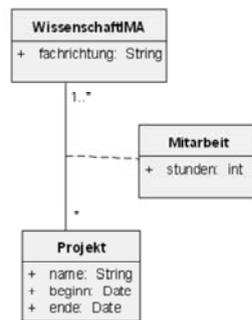


```
class Fakultät {
    ...
    private Institut i1, i2, ..., in;
    ...
    public Fakultät () {
        i1 = new Institut();
        i2 = new Institut();
        ...
        in = new Insitut();
    }
    ...
}
```

- Nun müssen die Operationen, die auf den Instituten durchgeführt werden können, durch die Fakultät zur Verfügung gestellt werden

Wir kommen zur Übersetzung der starken Aggregation. Wenn eine „Fakultät“ erzeugt wird, werden ihre „Institute“ innerhalb des Konstruktors ebenfalls erzeugt. Damit sieht man, dass eine Existenz-Abhängigkeit besteht und dass ein „Institut“ niemals alleine erzeugt werden kann, sondern nur als Teil einer „Fakultät“.

Übersetzung nach Java: Assoziationsklasse



```
import java.util.Hashtable;

class WissenschaftlMA {
    ...
    private Hashtable rProjekt;
        //Schlüssel: Projekt
        // Wert: Mitarbeit
    ...
}
```

- Die Assoziation wird mit Hilfe einer Hashtable abgebildet
- Ist die Assoziation nicht gerichtet, muss in der gegenüberliegenden Klasse ebenfalls eine Hashtable eingefügt werden

Ein bisschen schwieriger ist die Übersetzung von Assoziationsklassen. Wie löse ich denn die auf? Eine Möglichkeit eine Assoziationsklasse abzubilden ist ein Hashtable.

Ich habe hier einen wissenschaftlichen Mitarbeiter, der an mehreren Projekten arbeitet und an einem Projekt arbeiten auch mehrere wissenschaftliche Mitarbeiter. Der wissenschaftliche Mitarbeiter kann zwar an mehreren Projekten beteiligt sein, aber bei jedem Projekt nur einmal. Daher kann man die Beteiligungen in einer Hashtable des wissenschaftlichen Mitarbeiters anlegen, wobei der Schlüssel das Projekt ist, an dem er beteiligt ist und der Wert die Beteiligung.

Die Frage, die sich jetzt ergibt, lautet: Ist diese Überleitung immer möglich? Wann kann ich einen Hashtable nicht zum Abbilden einer Assoziationsklasse verwenden? Ich kann diese Überleitung nicht verwenden, wenn der wissenschaftliche Mitarbeiter an einem Projekt mehrmals beteiligt ist, denn der Schlüssel des Hashtable „Projekt“, muss ja eindeutig sein. Im Default-Fall ist das Ende einer Assoziation ja immer „unique“, wie wir kennen gelernt haben. Sobald ich ein Assoziationsende jedoch auf „non-unique“ setze, bedeutet das, der Mitarbeiter könnte auch in mehreren Projekten beteiligt sein und dann kann ich den Hashtable nicht mehr verwenden, da das Projekt nicht mehr als eindeutiger Schlüssel verwendet werden kann.

Übersetzung nach Java: Zusammenfassung

- Klassen werden nach Java-Klassen übersetzt
- Attribute und Operationen werden in Java als Instanzvariablen und Methoden dargestellt
- Klassenvariable und -operationen (unterstrichen im Klassendiagramm) werden mit dem Schlüsselwort `static` versehen
- Assoziationen werden mit Hilfe von Variablen ausgedrückt
 - für 1:1-Beziehungen reicht jeweils eine Variable vom Typ der verbundenen Klasse
 - für 1:n-Beziehungen braucht man Arrays, ArrayLists oder Ähnliches
 - Angabe von Navigationsrichtung (unidirektional) vereinfacht i.A. die Implementierung
 - Operationen zur Verwaltung der Assoziationen müssen eingefügt werden
- Einfachvererbung wird von Java direkt unterstützt (`extends`)
- Assoziationsklassen können mit Hashtables dargestellt werden

Hier haben wir noch die Zusammenfassung der Übersetzung des Klassendiagramms nach Java.

Klassen in UML werden auch als Klassen in Java übersetzt.

Attribute und Operationen werden in Java als Instanzvariablen und Operationen dargestellt.

Klassenvariable und -operationen sind im Klassendiagramm unterstrichen und werden mit dem Schlüsselwort „static“ versehen.

Assoziationen werden mit Hilfe von Variablen ausgedrückt:

Für eine 1:1-Beziehung reicht es jeweils eine Variable vom Typ der verbundenen Klasse, in der gegenüberliegenden Klasse aufzunehmen.

Für 1:n-Beziehungen brauche ich dann „Array“, „ArrayList“ oder ähnliches.

Die Angabe zur Navigationsrichtung ist entscheidend: Wenn ich nämlich nur in eine Richtung navigieren kann, dann muss ich nur auf der einen Seite wo keine Pfeilspitze ist ein Attribut aufnehmen, aber nicht auf der Seite der Pfeilspitze. Wenn aber auf beiden Seiten Pfeilspitzen sind, dann muss ich auf beiden Seiten ein Attribut aufnehmen. Letzteres gilt natürlich auch für den Best Practice Fall wo einfach eine Linie ohne Pfeilspitzen zwischen den Klassen besteht.

Operationen zur Verwaltung von Assoziationen müssen eingefügt werden. Das ist natürlich entscheidend wenn ich Assoziationen als „private“ definiere, um mittels Operationen auf diese Beziehungen zugreifen zu können und die Menge der beteiligten Objekte verwalten zu können.

Die Einfachvererbung wird mittels „extends“ in Java direkt unterstützt. Die Überleitung einer Mehrfachvererbung ist nicht Gegenstand dieser Vorlesung. Trotzdem befinden sich am Ende des Foliensatzes einige Folien zu diesem Thema, die ich jedoch nicht durchführen werde.

Das einzige, das jetzt ein bisschen schwieriger ist, ist die Überleitung von Assoziationsklassen. Hier ist der Vorschlag, Hashtables zu verwenden, wenn die Beziehungsenden „unique“ sind.

Datentypen in UML

- Datentypen können wie Klassen Attribute und Operationen haben
- Aber: Instanzen eines Datentyps haben keine Identität
 - Objekte: Instanzen einer Klasse
 - Werte: Instanzen eines Datentyps (z.B. Zahl 2)
- Notation: Klassensymbol mit Schlüsselwort «datatype»

- Beispiel:



- Arten von Datentypen:
 - Primitive Datentypen
 - Aufzählungstypen

Wir kommen schon fast zum Ende des Klassendiagramms. Was wir bisher übergangen haben, waren die **Datentypen**.

Objekte sind Instanzen einer Klasse und besitzen eine eindeutige Objekt-ID. Werte sind Instanzen eines Datentyps, ohne Objekt-ID. Das ist ein entscheidender Punkt. Die Definition eines Datentyps schaut im Prinzip aus wie die Notation einer Klasse, jedoch wird der Datentyp mit dem Schlüsselwort „datatype“, geschrieben in Spitzklammern <<datatype>>, versehen. UML-spezifisch ist, dass Datentypen selbst wieder Attribute und Operationen besitzen können. Das ist aber nur in den seltensten Fällen der Fall.

Hier in unserem Beispiel haben wir den Datentyp „Zeit“ und dieser besitzt die Attribute „stunde“ und „minute“, sowie die Operation „differenz“. Generell unterscheiden wir zwei wesentliche Arten von Datentypen, nämlich **primitive Datentypen** und **Aufzählungstypen**.

Arten von Datentypen: Primitive Datentypen

- Primitive Datentypen: Datentypen ohne innere Struktur
- Von UML vordefinierte primitive Datentypen:
 - Boolean
 - Integer
 - UnlimitedNatural
 - String
- Primitive Datentypen können auch selbst definiert werden:
 - wie beliebige Datentypen, nur mit dem Schlüsselwort «primitive»
- Primitive Datentypen können ebenfalls Operationen haben

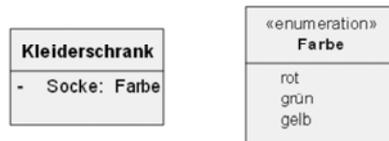
Es gibt in UML eine Reihe von **vordefinierten Datentypen**. Sie können sich aber auch Ihre eigenen Datentypen anlegen.

Das Problem ist, dass die Datentypen von Programmiersprache zu Programmiersprache unterschiedlich sind und auch oft ihre Wertebereiche unterschiedlich sind. Wenn ich mir anschau, was ist der Wertebereich von integer, long integer usw. so ist dieser nicht notwendiger Weise in allen Programmiersprachen gleich. Welchen Datentyp meine ich dann bei der Modellierung genau? Daher empfiehlt es sich, wenn Sie plattformspezifisch modellieren, d.h. Sie schon wissen in welcher Programmiersprache Sie den Code erzeugen werden, auch die Datentypen dieser Programmiersprache zu verwenden. Sie können sich die Datentypen einer Programmiersprache selbst erzeugen. Doch das nimmt Ihnen aber natürlich jedes vernünftige UML-Tool ab. Weil in jedem vernünftigen UML-Tool die Datentypen der Programmiersprachen, die durch dieses Tool unterstützt werden, schon vordefiniert sind.

Das Anlegen Ihrer eigenen **Datentypen** erfolgt mit Hilfe des Schlüsselwortes „primitive“. Dazu brauchen Sie wie beim Datentyp nur in Spitzklammern <<primitive >> schreiben, darunter den Namen des Datentyps und wenn Sie möchten und es ist sinnvoll, auch den Wertebereich des Datentyps. Natürlich könnten Ihre selbst angelegten primitiven Datentypen auch wieder Attribute und Operationen enthalten.

Arten von Datentypen: Aufzählungstypen

- Festlegung der Ausprägungsmenge per Aufzählung
- Notation: Klassensymbol mit Schlüsselwort «enumeration»
- Aufzählungstypen können Attribute und Operationen haben
- Mögliche Ausprägungen werden durch benutzerdefinierte Bezeichner (Literele) angegeben
- Beispiel:



Dann gibt es auch noch den **Aufzählungstyp**. Der Aufzählungstyp definiert die möglichen Werte. Aufzählungstyp, oder zu Englisch Enumerations, kennen Sie vielleicht auch aus Java.

In dem Beispiel haben wir die Enumeration „Farbe“ und „rot“, „grün“ und „gelb“ sind die erlaubten Werte.

Nutzen kann man die Enumeration dann für ein Attribut, z.B. für das Attribut „Socke“. Das Attribut „Socke“ besitzt den Typ „Farbe“ wobei eine Instanz von Farbe nur einen der Werte „rot“, „grün“ oder „gelb“ annehmen kann.

Inhalt

- Klassendiagramm
 - Klassen
 - Attribute und Operationen
 - Assoziationen
 - Schwache Aggregation
 - Starke Aggregation
 - Generalisierung
 - Zusammenfassendes Beispiel
 - Übersetzung nach Java
 - Datentypen in UML
- Objektdiagramm
- **Paketdiagramm**
- Abhängigkeiten



Als einen der letzten Punkte behandeln wir nun das **Paketdiagramm**, wobei ich jetzt einmal behaupte, dass kaum jemand freiwillig ein Paketdiagramm zeichnet wird. Paketdiagramme werden, wenn überhaupt, meist durch die UML-Tools auf Grund der Modell-Struktur automatisch erzeugt.

Paketdiagramm

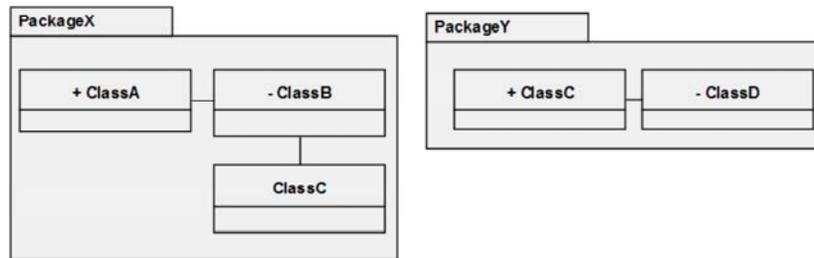
- UML-Abstraktionsmechanismus: Paket
- Modellelemente können höchstens **einem** Paket zugeordnet sein
- Partitionierungskriterien:
 - Funktionale Kohäsion
 - Informationskohäsion
 - Zugriffskontrolle
 - Verteilungsstruktur
 -
- Pakete bilden einen eigenen Namensraum
- Sichtbarkeit der Elemente kann definiert werden als »+« oder »-«

Wenn Sie ein UML-Modell haben, wollen sie nicht Modellelemente in einen Container werfen. Sie haben womöglich sehr viele Klassen. Wenn Sie außerdem Ihr System durchgängig modellieren, haben Sie zunächst ein Domänenmodell, das Ihnen einmal einen abstrakten Überblick über gewisse Klassen gibt, die in Ihrem System vorkommen. Später im Entwicklungsprozess werden Sie Klassen schon sehr implementierungsnah modellieren. Und diese beiden Klassendiagramme wollen sie eigentlich nicht verwechseln. Es könnte in beiden Modellen z.B. eine Klasse „Adresse“ vorkommen, die aber in ihren Details anders aussieht und daher anders definiert ist.

Aus der Programmierung kennt man dafür das Konzept des namespace oder Namensraum. Zwei Klassen mit demselben Namen unterscheiden sich, weil sie unterschiedlichen Namensräumen zugeordnet sind. Und dementsprechend habe ich hier das Konzept der Pakete bzw. Packages. Packages kann ich einem eindeutig definierten Namensraum zuordnen. Die Struktur von einem UML-Modell sieht aus, wie eine Folder-Struktur, wie Ihr Dateiodner, und so haben Sie es auch in den ganzen UML-Tools. Sie haben als Wurzel das Modell und darunter können Sie Folder also Packages anlegen und in diesen Packages befinden sich Klassen und andere Modellelemente. Packages können wieder Packages enthalten. Sie dienen zur Strukturierung. Sie werden Ihre einzelnen Modellelemente diesen Packages zuordnen, aber Sie werden kaum ein Paketdiagramm zeichnen. Das passiert automatisch. Wenn Sie z.B. eine Klasse anlegen, dann wird sie automatisch dem Package zugeordnet, in dem Sie sich gerade befinden.

Verwendung von Elementen anderer Pakete

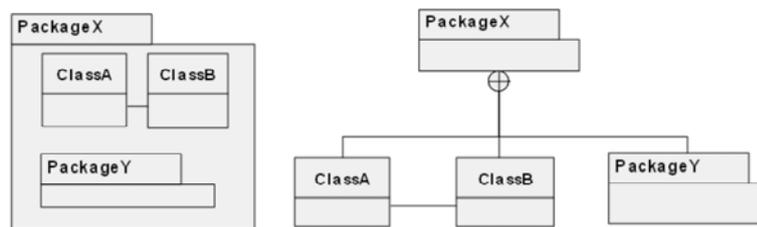
- Elemente eines Pakets benötigen Elemente eines anderen
- Qualifizierung dieser „externen“ Elemente
 - Zugriff über qualifizierten Namen
 - Nur auf öffentliche Elemente eines Pakets



Ein Diagramm befindet sich auch in einem Package. Sie können aber in diesem Diagramm nicht nur die Klassen, die sich im selben Package befinden, verwenden, sondern Sie können natürlich auch Klassen aus einem anderen Package verwenden. Natürlich nur die öffentlich sichtbaren Klassen dieser anderen Packages und der Zugriff erfolgt hier über einen qualifizierten Namen, d.h. Name der Klasse erweitert um den namespace.

Hierarchien von Paketen

- Pakete können geschachtelt werden
 - Semantik wird durch die Implementierungssprache bestimmt
 - Beliebige Tiefe
 - Paket-Hierarchie bildet einen Baum
- Zwei Darstellungsformen



Pakete können beliebig tief geschachtelt werden. Hier haben wir ein Beispiel. Es gibt das Package „X“. Das beinhaltet die Klasse „A“, die Klasse „B“ und das Sub-Package „Y“. Beide Darstellungsformen, die Sie auf dieser Folie sehen, sind äquivalent.

Import von Elementen und Paketen

- **Import einzelner Elemente**
 - Voraussetzung: Sichtbarkeit des Elements ist öffentlich
- **Import ganzer Pakete**
 - Äquivalent mit Element-Import aller öffentlich sichtbaren Elemente des importierten Pakets

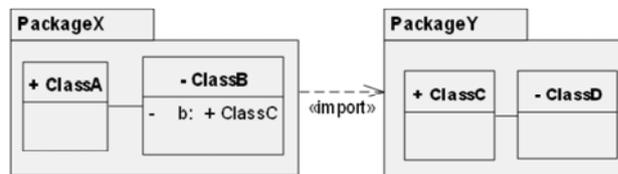
- **Sichtbarkeiten**
 - Beim Import kann die Sichtbarkeit der importierten Elemente und Pakete neu bestimmt werden
 - Sichtbarkeit nur öffentlich oder privat („+“ oder „-“)
 - «import»-Beziehungen für öffentliche Sichtbarkeit
 - «access»-Beziehungen für private Sichtbarkeit



Wenn wir von Namensräumen reden, ist das Konzept vom Importieren bzw. das Konzept vom Zugreifen auf ein anderes Package wichtig. Sie können einzelne Elemente importieren, dann müssen diese Elemente öffentlich sichtbar sein. Oder Sie können ganze Pakete importieren, das ist gleichwertig zum Import aller Elemente eines entsprechenden Packages. Beim Import kann die Sichtbarkeit der importierten Elemente und Pakete neu bestimmt werden. Wobei generell gesagt werden muss, dass die Sichtbarkeit nur öffentlich oder privat sein kann. Hierzu sehen wir uns auf weiteren Folien auch den Unterschied zwischen import- und access-Beziehungen an.

Import von Elementen und Paketen – «import» (1/2)

- Veränderung des Namensraums
 - Lädt die Namen des importierten Pakets in den Namensraum des Klienten
 - Ändert damit den Namensraum des Klienten
 - Qualifizierte Namen sind nicht mehr nötig

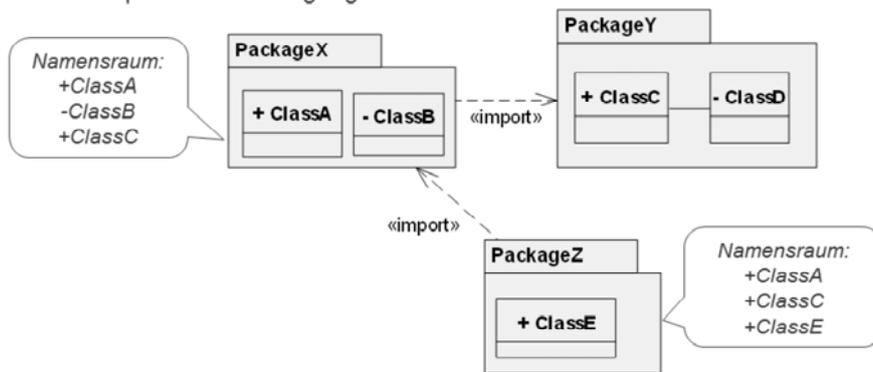


Beim **Import** von Elementen erfolgt auch eine Veränderung des Namensraums. Man legt die Namen des importierten Paktes in den Namensraum des Klienten. Man ändert damit auch den Namensraum des Klienten. Qualifizierte Namen sind nicht mehr nötig.

Import von Elementen und Paketen – «import» (2/2)

▪ Transitivität

- Die importierten Namen sind öffentlich und finden somit bei erneutem Import Berücksichtigung

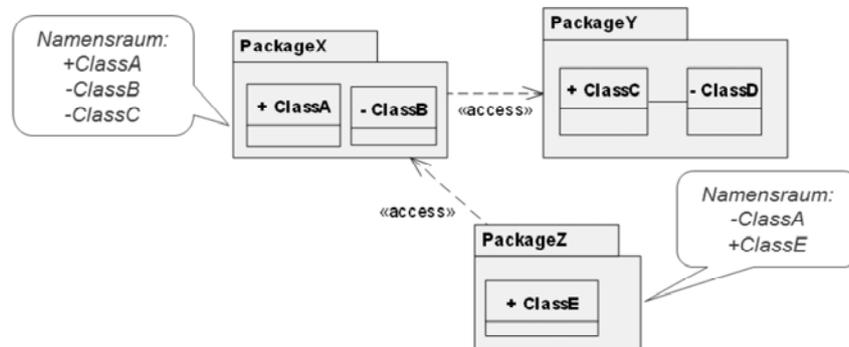


Ich erkläre den Import zum Abschluss an Hand von diesem Beispiel. Ich definiere hier, dass das Package „X“ das Package „Y“ importiert. Importiert werden alle Klassen, die nach außen hin sichtbar sind. Die Klassen werden importiert, nicht deren Attribute. Die Klassen sind ebenfalls als „public“ oder als „private“ definiert. Nach außen hin sichtbar sind nur die „public“ Klassen und diese werden sozusagen importiert. D.h. wenn „X“ „Y“ importiert, dann befinden sich in „X“ die Klasse „A“ und die Klasse „B“, weil diese schon nativ im Package „X“ enthalten sind und die „public“ sichtbare Klasse „C“ aus „Y“. D.h. in „X“ sehe ich „A“, „B“ und „C“.

Wenn das Package „Z“ nun „X“ importiert, importiert es wiederum alle öffentlich sichtbaren Klassen aus „X“. In „Z“ ist somit sichtbar die Klasse „E“ weil diese direkt als Inhalt von „Z“ definiert ist. Dann aus dem Package „X“ die Klasse „A“, weil diese öffentlich ist, nicht die Klasse „B“, weil diese nicht öffentlich ist und zusätzlich sehe ich in „Z“ noch die Klasse „C“, denn „X“ hat „C“ aus dem Package „Y“ importiert. Innerhalb von „X“ ist „C“ öffentlich sichtbar und wenn nun „Z“ das Paket „X“ importiert, ist auch innerhalb von „Z“ die Klasse „C“ öffentlich sichtbar.

Import von Elementen und Paketen – «access»

- Nicht-transitiv
- Änderung der Sichtbarkeit der importierten Elemente auf privat



Im Gegensatz dazu gibt es das Konzept „**access**“. „access“ unterscheidet sich von „import“ dadurch, dass nicht transitiv importiert wird, weil auf die Packages nur zugegriffen wird, sie aber nicht importiert werden. Greift das Package „X“ auf das Package „Y“ zu, so ist das Ergebnis das gleiche wie vorhin bei „import“. D.h. „A“, „B“ und „C“ sind innerhalb des Package „X“ sichtbar. Das Package „Z“ greift auf das Package „X“ zu. Dementsprechend ist in „Z“ das „E“ enthalten, weil es direkt in „Z“ definiert ist. Das aus „X“ öffentlich sichtbare „A“ ist ebenfalls enthalten in „Z“, aber das „C“ ist nun in „Z“ nicht mehr enthalten, weil es nicht transitiv importiert wird.

Inhalt

- Klassendiagramm
 - Klassen
 - Attribute und Operationen
 - Assoziationen
 - Schwache Aggregation
 - Starke Aggregation
 - Generalisierung
 - Zusammenfassendes Beispiel
 - Übersetzung nach Java
 - Datentypen in UML
- Objektdiagramm
- Paketdiagramm
- **Abhängigkeiten**

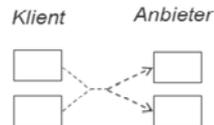
Das Konzept der Abhängigkeiten werden wir nur sehr oberflächlich durchnehmen. Zum detaillierten Verständnis ist es nämlich nötig, dass man schon einmal etwas größeres modelliert hat.

Abhängigkeiten

- Eine Abhängigkeit stellt eine **allgemeine Kopplung zweier Modellelemente** (Klassen, Interfaces, Pakete usw.) dar
 - Änderungen in einem Element (Anbieter) können Änderungen im abhängigen Element (Klient) nach sich ziehen
- Abhängigkeit zwischen genau zwei Modellelementen (Normalfall)



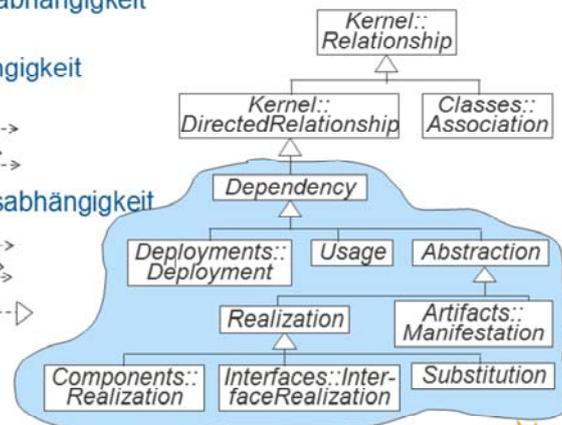
- Abhängigkeit zwischen zwei Mengen von Modellelementen



Die Abhängigkeit ist die schwächste Form einer Beziehung. Ich sage, das Modellelemente „A“ ist abhängig vom Modellelement „B“ und zeichne einen gestrichelten Pfeil ein. Die Aussage ist, wenn du in „B“ etwas änderst, wenn „A“ von „B“ abhängig ist, dann solltest du aufpassen, denn dann könnte das einen Effekt auf „A“ haben.

Arten von Abhängigkeiten

- **Dependency: Allgemeine Abhängigkeit** ----->
 - macht keine Aussage über genaue Art der Abhängigkeit
- **Deployment: Verteilungsabhängigkeit**
 «deploy»----->
- **Usage: Benutzungsabhängigkeit**
 «use»-----> «call»----->
 «instantiate»-----> «send»----->
 «create»----->
- **Abstraction: Abstraktionsabhängigkeit**
 «abstraction»-----> «trace»----->
 «refine»-----> «derive»----->
 Realisierungsabhängigkeit ----->▷
 «substitute»----->



Und dann gibt es eine Reihe von verschiedenen vordefinierten Abhängigkeiten: „use“, „instantiate“, „create“, „call“ usw., die alle eine besondere Bedeutung haben.

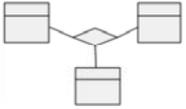
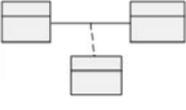
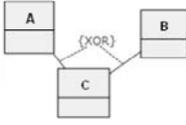
Bei der letzten Vorlesungseinheit zum Thema Anwendungsfälle werden wir auf jeden Fall noch solche Abhängigkeiten kennen lernen, denn zwischen Anwendungsfällen gibt es „include“- und „extends“-Beziehungen, die in Wirklichkeit solche Dependencies/Abhängigkeiten sind und keine Beziehungen. Ich hoffe, Sie verzeihen es mir dann, wenn ich zu diesem Zeitpunkt immer von „include“- und „extends“-Beziehungen spreche, obwohl es in Wirklichkeit Abhängigkeiten sind.

Basiselemente (1/3)

Name	Syntax	Beschreibung
Klasse		Beschreibung der Struktur und des Verhaltens einer Menge von Objekten
abstrakte Klasse		Klasse, die nicht instanziiert werden kann
Assoziation		Beziehung zwischen Klassen: keine Angabe über Nav.-r.; mit Navigationsrichtung; in eine Richtung nicht navigierbar.

Das war das Klassendiagramm, das mit Abstand wichtigste Diagramm von UML, in Kurzfassung und wir werden das Klassendiagramm in den Übungen noch weiter vertiefen. Auf den nächsten drei Folien findet sich eine Zusammenfassung der Basiselemente des Klassendiagramms.

Basiselemente (2/3)

Name	Syntax	Beschreibung
n-äre Assoziation		Beziehung zwischen n Klassen
Assoziationsklasse		nähere Beschreibung einer Assoziation
xor-Beziehung		Entweder steht Klasse A oder Klasse B in Beziehung zu C, nicht aber beide



Basiselemente (3/3)

Name	Syntax	Beschreibung
schwache Aggregation		"Teil-Ganzes"-Beziehung
starke Aggregation = Komposition		exklusive "Teil-Ganzes"- Beziehung
Generalisierung		Vererbungsbeziehung zwischen Klassen

Zusammenfassung

- Sie haben diese Lektion verstanden, wenn Sie wissen ...
- was der Unterschied zwischen einer Klasse und einem Objekt ist.
- was der Unterschied zwischen abstrakten und konkreten Klassen ist.
- was Attribute und Operationen einer Klasse sind und welche Eigenschaften diese haben können.
- dass Klassen durch Assoziationen miteinander verbunden werden.
- warum Assoziationen mit einer Multiplizität versehen werden.
- was Generalisierung ist und wann diese eingesetzt wird.
- was der Unterschied zwischen starker und schwacher Aggregation ist.
- wie Sie aus einer textuellen Angabe ein UML-Klassendiagramm erstellen.
- wie Sie die wichtigsten Elemente des Klassendiagramms in Java umsetzen können.
- wozu ein Meta-Modell benötigt wird.
- wie der Zusammenhang zwischen Klassen- und Objektdiagramm ist.
- was Interfaces sind.
- wozu Pakete eingesetzt werden.
- wie Abhängigkeiten in UML dargestellt werden können.



Ich hoffe, dass Ihnen alle Punkte, die auf dieser Folie gelistet sind, klar geworden sind. Hier endet nun die Einheit. In den Folien haben Sie noch weiterführende Themen zum Klassendiagramm, die jedoch allesamt nicht prüfungsrelevant sind.

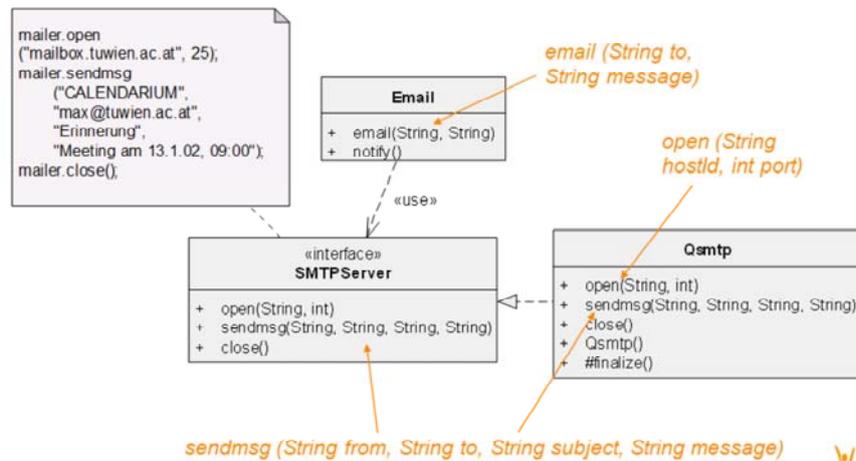
Anhang

Interface

- Ein Interface spezifiziert **gewünschtes Verhalten durch Zusammenfassung der Operationen**
 - einer Klasse
 - einer Komponente
 - eines Paketsund kann auch Attribute aufweisen
- **Unterschied zur abstrakten Klasse**
 - Abstrakte Klasse kann nur durch Subklassen realisiert werden, Interfaces durch beliebige Klassen
- Klassen, die ein **Interface realisieren - Anbieter** - können noch zusätzliche Operationen aufweisen
- Klassen, die ein **Interface benutzen - Klienten** - müssen nicht alle angebotenen Operationen tatsächlich nutzen

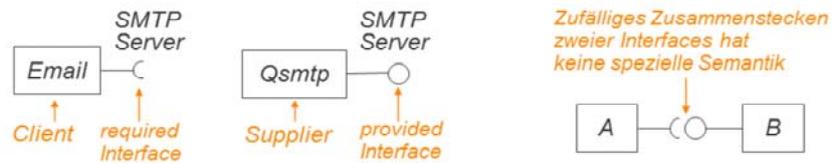
Interface: Beispiel CALENDARIUM (1/2)

- Notationsvariante 1:

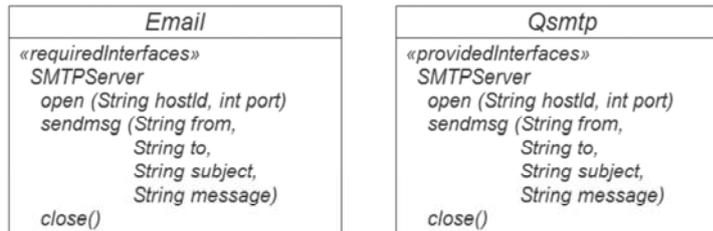


Interface: Beispiel CALENDARIUM (2/2)

- Notationsvariante 2:



- Notationsvariante 3:

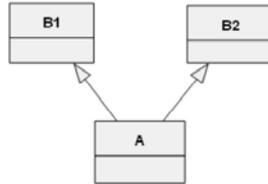


Interface: Vorteile

- Durch die Verwendung von Interfaces wird die **Vererbung von Implementierungen** von der **Vererbung von Interfaces** getrennt
 - Insbesondere **Frameworks** können größtenteils auf der Basis von Interfaces gebaut werden
- Eine **Klasse** kann als **Menge von Rollen** angesehen werden
 - Jedes **Interface** repräsentiert eine **Rolle**, die die Klasse spielt
 - Verschiedene **Klienten** verwenden nur jene Rollen, die für sie interessant sind
 - Mit Interfaces können **Sichten** auf eine Klasse für verschiedene Klienten realisiert werden
 - **Kopplung** wird reduziert, Flexibilität in Bezug auf Wartbarkeit und Erweiterbarkeit steigt

Übersetzung nach Java: Generalisierung – Mehrfachvererbung (1/3)

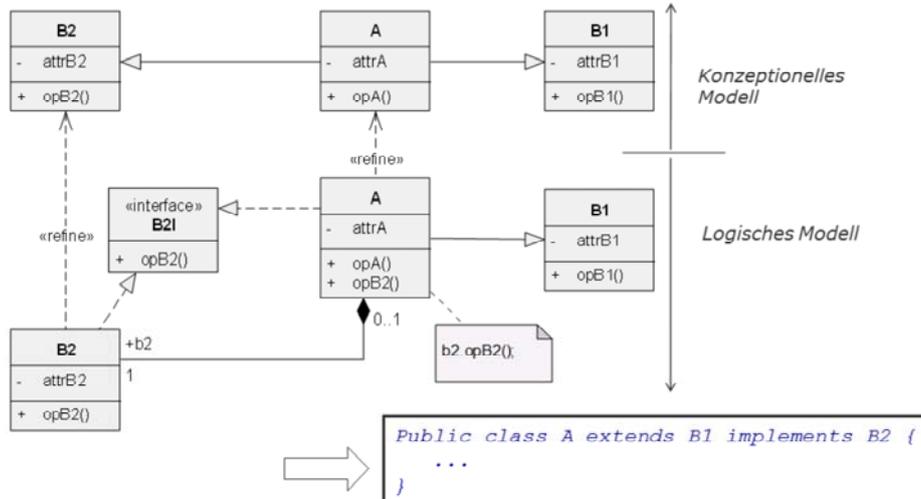
- Bei Simulation von Mehrfachvererbung zu beachten:
 - Spezifikationsvererbung („Muss“):
A erbt die nichtprivaten Schnittstellen von B1 und B2
 - Implementationsvererbung („Soll“):
A erbt die Implementation von B1 und B2
(Code-Wiederverwendung)



Übersetzung nach Java: Generalisierung – Mehrfachvererbung (2/3)

- Vorgangsweise bei n-fach-Vererbung:
 - 1 Mal: „echte“ Vererbung von Basisklasse B1
 - (n-1) Mal: simulierte Vererbung durch
 - Interface-Realisierung (Spezifikationsvererbung);
für die Basisklassen B2 ... Bn sind entsprechende Interfaces vorzusehen
(im Beispiel B2)
 - Komposition („Implementierungsvererbung“)
der „ehemaligen“ Basisklassen B2 ... Bn
(im Beispiel B2)

Übersetzung nach Java: Generalisierung – Mehrfachvererbung (3/3)



Metamodell

- Aufgabe: Entwicklung eines Tools zum Erstellen von Modellen (z.B. von Klassen- und Anwendungsfalldiagrammen)
- Problem: Wie stellen wir die Modelle generisch dar? (z.B. wie beschreiben wir, dass Klassen durch Assoziationen mit einander in Beziehung stehen können?)

Wir brauchen ein **Modell von den Modellen**
= **METAMODELL**

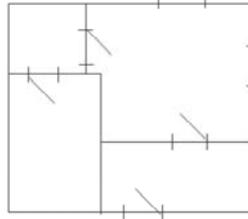
- auch UML besitzt ein Metamodell: die **Superstruktur**
 - Definition von Sprachkonzepten und deren Grammatiken für die Spezifikation von Modellen
 - UML kann mit einer Teilmenge von UML spezifiziert werden

Beispiel: Metamodell der Architektur

Reales Objekt

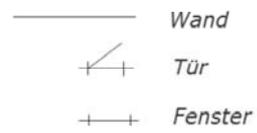


Modell (Plan)



Metamodell

Konstrukte (Objekttypen):



Attribute:

- Ein Fenster hat Breite und Höhe
- Eine Wand hat eine Stärke

Regeln:

- Eine Tür grenzt links & rechts an eine Wand
- Fenster sind an Außenwänden

Erweiterung von UML

- Manchmal sind die von UML zur Verfügung gestellten Sprachkonstrukte unzureichend
 - ⇒ Erweiterung von UML ist notwendig

Varianten:

- 1.) Neues Metamodell
 - 2.) Erweiterung des UML-Metamodells
 - a) Unkontrolliert
 - b) Mit vorgegebenen Erweiterungsmechanismen:
 - Stereotype
 - Tagged Value
 - Constraint
- Profile stellen den Hauptmechanismus dar, um UML zu erweitern
 - Technologische Erweiterungen (z.B. J2EE, .NET)
 - Domänenerweiterungen (z.B. Echtzeitanwendungen, EAI, Telekommunikation, Luftfahrt)

Stereotype

- Jedes Modellelement kann mit einem Stereotyp weiter klassifiziert werden («...» in der Nähe des Namens)
- Anstelle einer Erweiterung der Metaklassenhierarchie:
 - Repräsentieren zur Modellierungszeit eingeführte Erweiterungen von Metamodellelementen
 - Dienen einem spezifischen Zweck (z.B. «hardware», «software»)
 - Stellen eine Brücke zwischen Modell und Metamodell dar (befinden sich im Modell, gehören zum Metamodell)
- Selbstdefinierte Symbole sind zulässig
- Beispiel:



- Vordefinierte Stereotype: «file», «executable», «send», «entity», ...

Tagged Values

- Schlüssel/Wert-Paar (tagged value)
- Belegung der Attribute eines Stereotyps mit Werten
- Notation: {tag = value}
 - Vordefinierte Paare dienen i.A. zur Angabe von Metamodellattributen, z.B.
 - {persistence=transitory/persistent}
 - {isAbstract=true/false}
 - Statt {isX=true} kann einfach {X} geschrieben werden
 - Benutzerdefinierte Paare werden innerhalb von UML nicht interpretiert, z.B.
 - {author="Martin"}