Datenbanken II A: DB-Entwurf

Chapter 11: UML Class Diagrams I

Prof. Dr. Stefan Brass

Martin-Luther-Universität Halle-Wittenberg

Wintersemester 2020/21

http://www.informatik.uni-halle.de/~brass/dd20/

Objectives

After completing this chapter, you should be able to:

- read and write UML class diagrams.
- translate ER-schemas into UML class diagrams and vice versa.
- translate a UML class diagram into a relational database schema (as far as possible).
- explain differences between the object-oriented and the classical relational approach to database design.

  Especially with regard to operations and keys. What are the implementation options for operations in a RDBMS?

Contents

1 History and Importance of UML

2 Classes, Attributes

3 Associations

What is UML? (1)

- “ The Unified Modeling Language (UML) is a general-purpose visual modeling language that is used to specify, visualize, construct, and document the artifacts of a software system. It captures decisions and understanding about systems that must be constructed.”

  [Rumbaugh et.al., The UML Reference Manual, 1999]

- The UML gives you a standard way to write a system’s blueprints, covering conceptual things, such as business processes and system functions, as well as concrete things, such as classes written in a specific programming language, database schemas, and reusable software components.”

  [Booch et.al., The UML User Guide, 1999]

What is UML? (2)

- “The UML, in its current state, defines a notation and a meta-model. The notation is the graphical stuff you see in models; it is the syntax of the modeling language.”

  [Fowler/Scott, UML Distilled, Second Edition, 2000]

- “The UML is a modeling language, not a method. The UML has no notion of a process, which is an important part of a method.”

  [Fowler/Scott, UML Distilled, Second Edition, 2000]

History of UML (1)

Object-Oriented Programming Languages:

- Simula-67 (1965–1970) is generally called the first object-oriented language.

  Simula was developed by Nygaad and Dahl at the Norwegian Computing Center.

- Smalltalk is the classical object-oriented programming language. It was developed at XEROX PARC in the 1970s and became widespread in the 1980s.

  The first version was developed by Alan Kay and others in 1972. The book "Smalltalk-80: The Language and Its Implementation" by Adele Goldberg and David Robson appeared 1983.

History of UML (2)

Object-Oriented Programming Languages, Continued:

- C++ was introduced in 1984, but further developed during the 1980s and 1990s.

  In 1979–1980 Bjarne Stroustrup developed "C with Classes" at the Computer Science Research Center of Bell Laboritories in Murray Hill. During 1982–1984 it was redesigned and called C++, 1986 appeared the book "The C++ Programming Language". However, many features were still added later. The ANSI/ISO C++ standardization started in 1989 and the standard was finally approved in 1999.

- Eiffel: Developed 1985–1986, the book by Bertrand Meyer appeared 1988. "Design by Contract".
- Java: Introduced in 1995 (by SUN Microsystems).

History of UML (3)

Development Methods for Traditional Languages:

- E.g. Structured Analysis and Structured Design, a development method for traditional programming languages, was published by Edward Yourdon and Larry L. Constantine in 1979.
- Development methods became widespread in the 1980s.

History of UML (4)

Object-Oriented Design Methods / Influential Books:

- Shlaer/Mellor (1988/1989)
- CRC: Wirfs-Brock/Wilkerson/Wiener (1990/91).
- Coad/Yourdon (1991)
- Booch [Rational Software Corporation] (1991)
- OMT: Rumbaugh/Blaha/Premerlani/Eddy/ Lorensen (1991).
- Martin/Odell (1992).
- OOSE: Jakobson et.al. [Objectory] (1992).

History of UML (5)

- “The number of object-oriented methods increased from fewer than 10 to more than 50 during the period between 1989 and 1994.”

  [Booch et.al., The UML User Guide, 1999]

- “. . . in 1994, the methods scene was pretty split and competitive. Each of the aforementioned authors was now informally leading a group of practitioners who liked his ideas.”

  [Fowler/Scott, UML Distilled, Second Edition, 2000]

- All methods/design languages were relatively similar, but each had strengths and weaknesses.

- In addition, similar things were often expressed in different notation.

History of UML (6)

- In October 1994, James Rumbaugh joined Grady Booch at Rational. Their goal was to unify the Booch and OMT methods.

  > "Grady and Jim proclaimed that 'the methods war is over — we won,' basically declaring that they were going to achieve standardization the Microsoft way." [Fowler/Scott, UML Distilled, Second Edition, 2000]

- In October 1995, the version 0.8 draft of the "Unified Method" was released.
- In Fall 1995, Rational bought Objectory and Ivar Jacobson joined the team working on UML.
- In June 1996, UML version 0.9 was published.
- In 1996, the Object Management Group (OMG) issued a request for a standard object-oriented modeling language.

History of UML (7)

- Rational formed a UML consortium (including, e.g., DEC, HP, IBM, Microsoft, Oracle, TI) that developed UML 1.0, offered for standardization to the OMG in January 1997.
- Until July/September 1997, most of the proposals for the OMG call were merged in the UML 1.1.
- UML 1.1 was adoped by the OMG on November 14, 1997.
- UML 1.3 was formally published in March 2000.
- UML 1.4.2 became ISO/IEC standard 19501:2005.
- UML 1.5 was published in March 2003.

History of UML (8)

- The UML 2.0 specification was separated into:
    - UML 2.0 infrastructure: Defines basic and commonly used elements (e.g., class, association, multiplicity), from which other model elements can be derived.
    - UML 2.0 superstructure: Defines further constructs, e.g. use cases, activities, statecharts.
    - UML 2.0 OCL (Object Constraint Language).
- The development of UML 2.0 was done from 2000 to 2006.
- The current version is UML 2.5.1 from December 2017.

Some Critical Remarks (1)

- It is visible in the UML that it is a monster language designed by a committee.

  It seems that everything is in what one committee member wanted in, so it is a very large language. Programming languages like PL/1 and Ada basically failed because of this.

- At least in the beginning, UML was not precisely defined.

  In some questions, the UML User Guide and the UML Reference Manual (both 1999, both from the “three amigos”) directly contradict each other. Other important questions about the exact meaning of certain constructs are simply not answered. Some software engineers think that UML is an acronym for “The Undefined Modeling Language”.

Some Critical Remarks (2)

- “UML is far from being new. With respect to syntax it just reinvents many . . . concepts and introduces new names for them. With respect to semantics it does not present precise semantic definitions. If these were added, the limitations of the expressiveness of the UML [would] become apparent.”

  [Klaus-Dieter Schewe: UML: A Modern Dinosaur? — A Critical Analysis of the Unified Modeling Language. Proc. 10th European-Japanese Conference on Information Modelling and Knowledge Bases, 2000] See also [http://www.dbdebunk.com/page/page/622530.htm].

Future (1)

- After the past experience and all this work on standardization, nobody seems to want another "methods war". At least not only about notation.
- In addition, UML has several extension mechanisms that allow to introduce new concepts in the notation.
- So it seems that UML is the future and all the direct successors to it (like OMT) are dead.

Future (2)

- ER-Diagrams are no direct successor to UML, and the DB community is relatively distinct from the OO design community.

  Already object-oriented databases did not have the commercial impact that was expected and several OODBMS vendors moved to different fields.

- If you start today a large software project without using an object-oriented language and UML, people find you strange. If you use an RDBMS and ER-diagrams, this is still acceptable.

  Advantage of UML: One language for software and DB.

Future (3)

- Some DB Design tools (e.g. Power Designer) introduce support for UML, but they continue to support ER-diagrams.

  And probably for quite some time. The support for ER-diagrams might still be better than that for UML. Oracle added UML to Oracle Designer (in a separate program: ODD) and removed it again.

- "Conceptually, an object does not need a key or other mechanism to identify itself, and such mechanisms should not be included in models."

  [UML Reference Manual, p. 294]

- But keys *are* important in DB design.

Contents

1 History and Importance of UML

2 Classes, Attributes

3 Associations

Classes (1)

- “Classes are the most important building block of any object-oriented system.”
- “A class is a set of objects that share the same attributes, operations, relationships, and semantics”.
- “You use classes to capture the vocabulary of the system you are developing.”

  All three cited from the UML User Guide [Booch et al, 1999].

- So a class is similar to an entity type, only operations are added.

  The meaning of operations for databases is discussed below.

Classes (2)

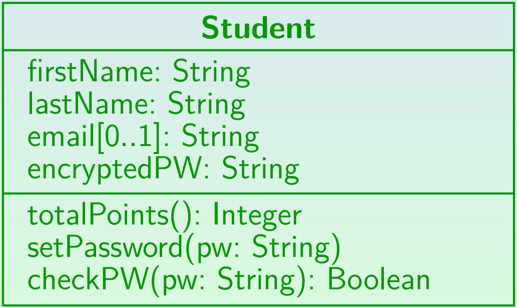
- One can use class diagrams in UML simply like a different syntax for ER-diagrams.
- However, the UML can be used to model the entire database application system.

  I.e. not only the database design, but also the software.
- So classes describe not necessarily persistent objects that might ultimately be stored as rows in a relational table.
- UML classes can also describe transient objects, e.g. C++ or Java objects that exist only for the duration of a program execution.

  Actually, the mapping to an object-oriented programming language or an OODB is more direct than to a relational database. But in this course, our intention is mainly to translate a UML class diagram into a relational DB schema.

Classes (3)

- A class is symbolized by a rectangle with normally three “compartments” (sections) that contain the class name, the attributes, and the operations:



Student

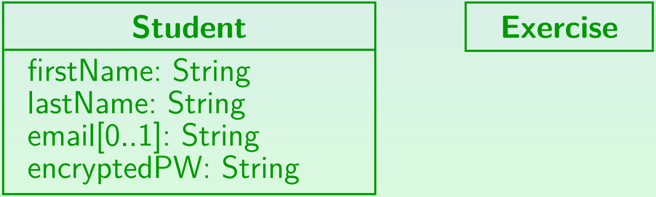
firstName: String
lastName: String
email[0..1]: String
encryptedPW: String

totalPoints(): Integer
setPassword(pw: String)
checkPW(pw: String): Boolean

Classes (4)

- Either or both of the middle and bottom compartment may be suppressed, i.e. it is possible to show only attributes, only operations, or none of the two.

  Operations always have a parameter list (which may be empty), so if the rectangle has only two compartments, one can tell from the () whether operations or attributes are shown.



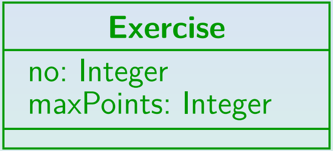
Student

firstName: String
lastName: String
email[0..1]: String
encryptedPW: String

Exercise

Classes (5)

- One often sees empty compartments, e.g.



Exercise

no: Integer
maxPoints: Integer

- This means that the class has no operations.

  Unless some kind of filtering is in effect, e.g. only public operations (see below) are shown.

- But some authors are so used to the three compartments that they still show the delimiting lines even if they do not show attributes or operations.

Classes (6)

Style guidelines (suggestions by the UML designers):

- One normally uses a noun or noun phrase (singular form) as class names.

  Class names should begin with an uppercase letter. One capitalizes the first letter of every word. The class name is printed centered and in boldface. Abstract classes (see next slide) are shown in italics.

- Attribute names are normally nouns/noun phrases.
- Operation names are usually verbs/verb phrases.

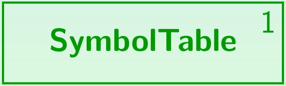
  Attribute and operation names start with a lowercase letter, but have the first letter of every following word capitalized. They are shown in normal font and left justified.

Classes (7)

- Abstract classes cannot have any direct instances (i.e. objects of that class cannot exist).

  Abstract classes can be useful to define a common interface, subclasses of this class can have instances.

- One can also define a multiplicity of a class, i.e. the number of instances (objects) of that class. It is written in the upper-right corner of the class rectangle:



SymbolTable

1

Extension Mechanisms (1)

- Besides the three predefined compartments (class name, attributes, operations), a class rectangle can have further user-defined named compartments.

  One application in the database context would be a compartment for triggers.

- One such user-defined compartment is already defined in the UML specification: Responsibilities.

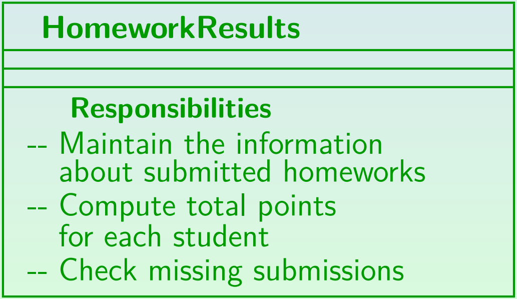
- Responsibilities explain the purpose of a class on a higher level than attributes and operations.

  "A responsibility is a contract or an obligation of a class." [UML User Guide, p. 53]

Extension Mechanisms (2)

- The responsibility compartment contains free text.

The responsibilities are usually written as itemized list. If a class has more than five responsibilities, it is probably too complicated.



HomeworkResults

Responsibilities

-- Maintain the information
about submitted homeworks
-- Compute total points
for each student
-- Check missing submissions

Extension Mechanisms (3)

- Stereotypes modify/redefine the semantics of existing UML constructs.

  So in effect one can add new constructs to the UML. Stereotypes correspond to creating a new subclass in the UML meta model.

- For instance, one can use the normal class notation, but add the stereotype "`utility`". This means that
  - the attributes of the class are global variables,
  - the operations are global functions.

  In this way, existing non-object-oriented library modules can be included.

Extension Mechanisms (4)

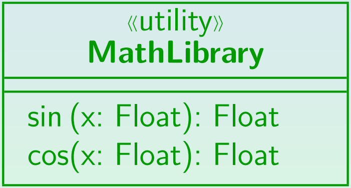
- The four standard stereotypes for classes are:
  - `metaclass`
  - `powertype`
  - `stereotype`
  - `utility`
- In addition, the following standard steoreotypes or keywords apply to classes (continued on next slide):
  - `interface`
  - `type`

Extension Mechanisms (5)

- Standard stereotypes or keywords, continued:
  - `implementationClass`
  - `actor`
  - `exception`
  - `signal`
  - `process`
  - `thread`
- However, the power of stereotypes is that the UML user can introduce new ones.

Extension Mechanisms (6)

- Steoreotypes are enclosed in « and » and are written in front of (or above) the declaration of the element that is modified:



«utility»
MathLibrary

sin (x: Float): Float
cos(x: Float): Float

- Instead of explicitly showing the stereotype name, one can also define new icons for the modified constructs.

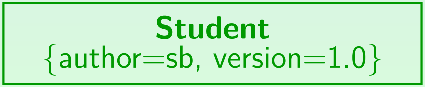
Extension Mechanisms (7)

- Every element in a specification (e.g. a class, an attribute) has certain properties.

- The set of these properties is user-extensible.

  Whereas stereotypes correspond to adding a subclass to the meta-model of UML, such "tagged values" in effect add an attribute.

- Additional properties are shown in a property list/as tagged values behind or below the element declaration enclosed in { and }.



Student
{author=sb, version=1.0}

Extension Mechanisms (8)

- The standard tagged values for classes are
  - `documentation` (any text),
  - `location` (e.g. client or server),
  - `persistence`, and
  - `semantics`.
- If needed, one can mark database classes with

  `{persistence=persistent}` or just `{persistent}`

  and program classes with

  `{persistence=transient}` or just `{transient}`.

Extension Mechanisms (9)

- As already shown in the example, if a property is of an enumerated type and an enumeration value implies a unique property name, if suffices to put that value in the property list.
- Of course, `{persistent}` and `{transient}` should only be used if the same diagram shows both kinds of classes. Otherwise it would overload the diagram.

Attributes (1)

- "An attribute represents some property of the thing you are modeling that is shared by all objects of that class."

  [Booch et.al.: UML User Guide, 1999, p. 50]

- "An attribute is the description of a named slot of a specified type in a class, each object of the class separately holds a value of the type."

  [Rumbaugh et.al.: UML Reference Manual, 1999, p. 166]

Attributes (2)

Attribute Scope:

- Attributes can have
  - class scope (class attributes, static members), or
  - instance scope (normal attributes).
- Attributes of class scope have only one value for the entire class (even if the class has no objects).

  Attributes of instance scope have one value for each object/instance of the class.
- Attributes of class scope are marked by underlining.

Attributes (3)

Attribute Visibility:

- Attribute visibility defines which classes can directly access the attribute (in their operations).
- There are three options:
  - public (+): The attribute is visible to any class that can see the class containing the attribute.
  - package (~): Visible to all classes of the package.
  - protected (#): Visible to the class itself and its subclasses.
  - private (−): Visible only to the class itself.

Attributes (4)

Multiple-Valued Attributes:

- UML permits multiple-valued attributes, i.e. sets or arrays. Example multiplicity specifications are:
  - `[0..1]`: Zero or one values.

    This corresponds to an attribute that can be null.
  - `[1..*]`: A set with at least one element.

    There is no upper bound on the number of elements. When translating a class with such an attribute into relations, one would create an extra table for this attribute. Exercise: Consider a class for web pages, where each web page has an URL, a title, and a set of keywords/search terms. Model this in UML and in the RM.
  - `[3 ordered]`: An array with three elements.

    The default is "`unordered`", i.e. a set.

Attributes (5)

Attribute Declaration:

- A full attribute declaration consists of:
  - Visibility: +, ~, #, - (see above).
  - The name of the attribute.
  - The multiplicity (array/set), e.g. [0..1], [3].
  - A colon ":" and the type of the attribute.
  - An equals sign "=" and the initial value of the attribute.
- Of this, everything except the name is optional.

Attributes (6)

- Example:

```
+ProgramOfStudy [0..2]: String = "MIS"
```

- In addition, the standard UML extension mechanisms apply:
  - In front of an attribute declaration, a stereotype can be specified (enclosed in « and »).
  - After the attribute declaration, a property string (enclosed in { and }) can be added.
  - In the property string, one can specify, e.g., the following values:
    - `changeable` (the default),
    - `frozen` (cannot be changed after object is initialized),
    - `addOnly` (for attributes with multiplicity > 1).

Constraints (1)

- “With constraints, you can add new semantics or change existing rules.”

  [Booch et.al.: The UML User Guide, 1999, page 82]

- This is not quite the usual notion of a constraint: In databases, a constraint can only restrict DB states.

  This shows again that database people and UML people do not speak the same language. To be fair, the UML reference manual states “A constraint is a semantic condition or restriction expressed as a liguistic statement in some textual language.”

  [Rumbaugh et.al.: The UML Reference Manual, 1999, page 235]

- Constraints are one of the three UML extension mechanisms (besides stereotypes, tagged values).

Constraints (2)

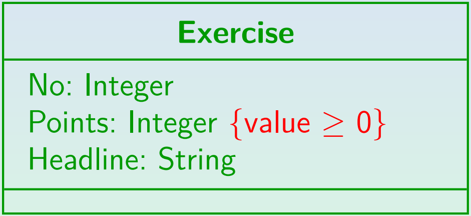
- Constraints are enclosed in { and } and written near to the element to which they apply.

  A constraint can be connected with dashed lines to the diagram elements to which it applies (if it is not clear from its position). It can be written into a note box, or simply on the diagram background.

- Constraints can be written
  - as free-form text,
  - in a formal logical language, especially OCL: UML's Object Constraint Language,
  - in a programming language.
  - as predefined name/abbreviation.

Constraints (3)

- Example (Restriction of an attribute):



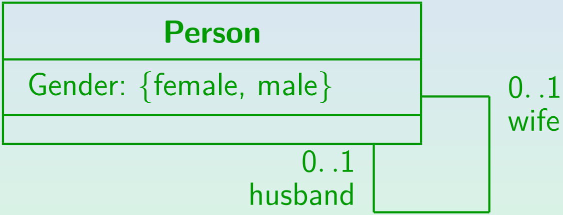
Exercise

No: Integer
Points: Integer $\{\text{value} \geq 0\}$
Headline: String

- If a constraint appears as an item of its own in the attribute list, it applies to all following attributes until it is explicitly cancelled.

Constraints (4)

- Example (using OCL for a relationship):



{self.wife.gender = female and
self.husband.gender = male}

[Booch et al., UML User Guide, 1999, p. 82]

Derived Attributes (1)

- Attributes are derived if can be computed from other attributes.

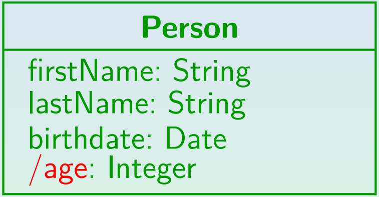
  The derivation formula can be shown as a constraint.

- Derived attributes should normally not be stored in the database, because they are redundant.

  Therefore, they seldom appear in conceptual database schemas (they do not give any additional information). However, if they are important concepts in the application domain, they can be included if they are explicitly marked as “derived”. Then they will typically be translated into a view, not into a stored column. There is no real difference between a derived attribute and a query operation.

Derived Attributes (2)

- Derived attributes are marked by putting a slash "/" in front of their name:



Person

firstName: String
lastName: String
birthdate: Date
/age: Integer

- Also other model elements can be derived. They are marked in the same way.

  E.g. relationships (called "associations" in UML, see below) might be computable from other relationships and/or attributes.

Keys

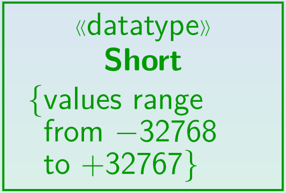
- UML has no built-in notion of keys.

  The idea is that objects automatically have an object identity, i.e. a surrogate key (an automatically generated number). However, at least externally objects must be identified in user input. Internal numbers/addresses are difficult for this purpose.

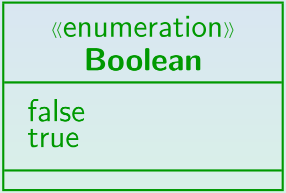
- One can extend UML in order to add keys. Several proposals exist, one is to add "`{oid}`" (or "`{pk}`") as property list to the primary key attributes.

  One would use "`{oid1}`" (or "`{ak1}`") for the attributes of the first alternative key, and so on. Some proposals also permit to define the sequence of the attributes in composed keys.

Specification of Data Types



«datatype»
Short
{values range
from −32768
to +32767}

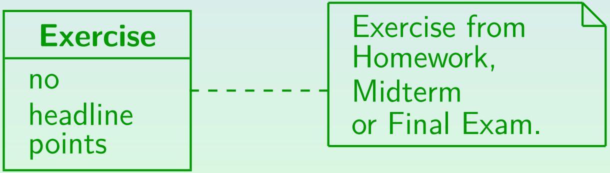


«enumeration»
Boolean

false
true

Annotations/Comments

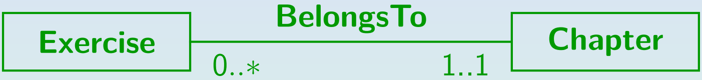
- A note can contain a comment, a constraint, or a method.
- It is shown in a dog-eared rectangle with its upper-right corner bent over:



Contents

Associations (1)

- Relationships are called "associations" in UML:



- Note that the cardinalities are written on the opposite side of the standard (min,max)-cardinalities:
  - Each exercise belongs to exactly one chapter.
  - A chapter can contain any number of exercises.
- Cardinalities are called multiplicities in UML.

Associations (2)

- Of course, in a relational database, associations are implemented as usual:
    - For a one-to-many relationship, one adds the key of the "one" side (Chapter) as a foreign key to the "many" side (Exercise).
    - For a many-to-many relationship, one constructs an "intersection table".
- But in order to understand UML better, it is also important to look at the implementation in object-oriented programming languages or databases.

Associations (3)

- In OODBs, associations are usually implemented by pointers that are the inverse of each other:

```
class Chapter (extent chapters)
{    attribute unsigned short number;
     attribute string title;
     relationship set<Exercise> contains
                         inverse Exercise::belongs_to;
};
class Exercise (extent exercises)
{    ...;
     relationship Chapter belongs_to
                         inverse Chapter::contains;
};
```

Associations (4)

- The example above is in the ODMG ODL.

  The Object Data Management Group has defined a standard for object-oriented database systems. The Object Definition Language is used for defining database schemas.

- In order to traverse the relationship efficiently in both directions, pointers are needed in both participating classes.

- If the system knows the inverse relationship, it can ensure the consistency.

  In particular, when an object is deleted, dangling pointers can be avoided, since the system knows which other objects contain pointers to the deleted object.

Multiplicity (1)

- A multiplicity specification consists of a comma-separated list of intervals, e.g. `0..2,5..6` means that the following numbers are possible: `0,1,2,5,6`.
- An interval consisting only of a single number can be denoted by that number, e.g. `1` is an abbreviation for the interval `1..1`.
- "∗" denotes an unbounded number, e.g. `0..*` is the most general interval (any number).
- `0..*` can be shortend to `*`.

Multiplicity (2)

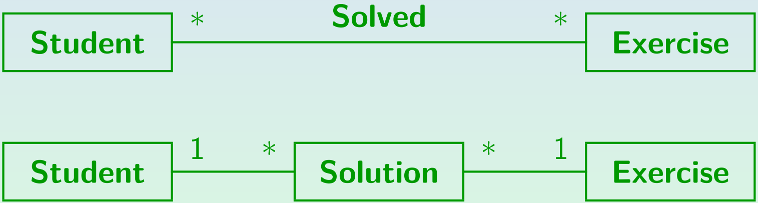
- The multiplicity specification near an entity type E_1 counts how many entities of this type can be related to a single entity of the other type E_2.

  The other notation counts the number of outgoing edges from a single entity of type E_1.

- The advantage of the UML notation is that the multiplicities for one-to-many relationships are as expected:
  - `1` (or `0..1`) on the "one" side and
  - `*` (or `1..*`) on the "many" side.

Multiplicity (3)

- The disadvantage of this notation is that if entities are introduced for many-to-many relationships, multiplicities must be moved around:

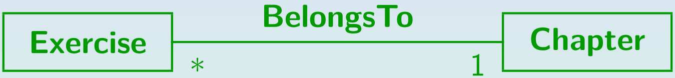


- With the standard (min,max)-notation this does not happen.

Because the number of outgoing edges does not change in this transformation.

Multiplicity (4)

- Another disadvantage is that if associations are implemented by pointers (as usual in object-oriented languages), the multiplicity is on the opposite side:



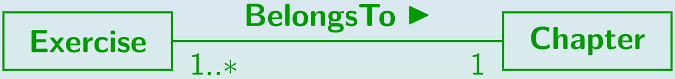
- Here, each Exercise object contains a single pointer to a Chapter object.

  One can get of course used to the UML notation and look to the other side. The multiplicity is near to the type of the pointer, which might be considered an advantage.

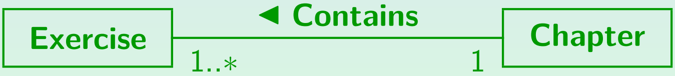
- But each Chapter object contains a set of pointers to Exercise objects (if this direction is supported).

Reading Direction

- One can use the symbol “►” to make the direction of the name clear (this is optional):



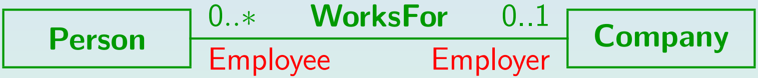
- Also ◄ ▲ ▼ can be used:



- Of course, it is best to choose names that read from left to right and from top to bottom.

Role Names (1)

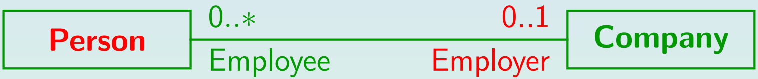
- Instead of or in addition to association names, one can also use role names:



- Here the person has the role of an employee in the association, and the company has the role of an employer.
- In other associations, objects of the two classes can play different roles (e.g., customer and contractor).

Role Names (2)

- The role names on the opposite site can often be used as attribute names for the pointers or foreign keys.

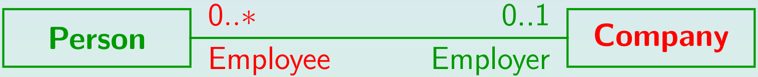


- In the example, "PERSONS" would have a foreign key (or pointer attribute) called "EMPLOYER".

It can contain 0 or 1 key values (pointers/addresses) of companies. In relational databases, this means that the attribute can be null.

Role Names (3)

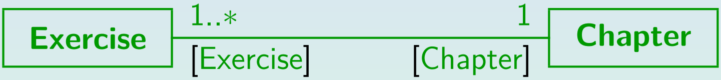
- If necessary, the “Company” class would have an attribute “Employees” that is a set of pointers to “Person” objects.



- Of course, in relational databases this is not necessary because with the foreign key on the “Person” side, the join can also find employees for a given company.

Role Names (4)

- Often, the class name itself can be used as role names. Then it is not necessary to add an explicit role name (actually, it is difficult to invent one).



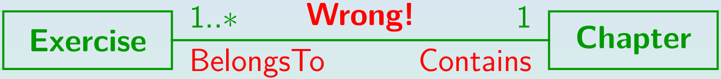
Note: The brackets "[. . .]" are not UML notation. They are intended to indicate that it does not matter wether these role names are explicitly written or not: They are the default.

- Then the table/class "Exercises" would have a foreign key (pointer attribute) "Chapter".

Conversely, "Chapter" might have a set-valued attribute "Exercises".

Role Names (5)

- The names used in the Barker Notation on both ends of the relationship are not role names in the sense of UML:



- UML tools would add a foreign key/pointer attribute “Contains” to the table/class “Exercises”.

  I.e. just the wrong way around.

- The names in the Barker Notation are really association names for both directions, not role names.

Uniqueness of Names

- Names of classes and associations must be unique.

  It is not even allowed to have an association and a class with the same name (since there are association classes, see below). UML has packages, and the uniqueness is only required within each package.

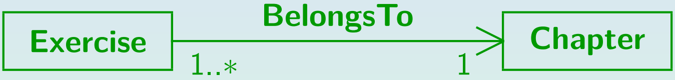
- Role names (labels of association ends) must be unique within the association (each end must have a different name) and within the connected class.

  A role may not have the same name as an attribute (since associations are typically implemented by pointer attributes).

- If there is only one connection between the two classes, it is possible to have neither association name nor role names.

Navigability (1)

- In UML, it is possible to specify that an association will be traversed only in one direction:



- Then Exercise objects would contain a pointer to the Chapter to which they belong, but there would be no inverse pointer.

  Even with the pointer implementation it might be possible to find exercises for one chapter (e.g. if there is a linked list of all exercise objects in the system). So the arrow only specifies in which direction an efficient traversal is possible.

Navigability (2)

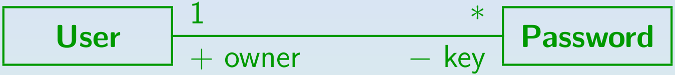
- Without inverse pointers, it might be difficult to ensure that when a Chapter object is deleted, the corresponding Exercise objects are deleted, too.

  An important reason for having pointers in both directions is to avoid dangling pointers. OODBMS can do this automatically.

- For a relational databases, the navigability specification is not important: Joins are always both ways.
- But for programs written e.g. in C++, one seldom has pointers in both directions, thus there the arrow will be often used.

Visibility

- Since relationships are implemented by attributes/operations, it possible to specify a visibility at the association ends:



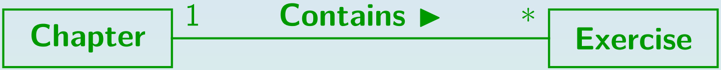
[Example from Booch et.al: UML User Guide, 1999, p. 145]

- This means that everybody who has access to a Password object, can navigate from there to the corresponding User.
- However, only operations of the User class can follow the link to the passwords.
- Thus the visibility is denoted at the opposite end of the association (the end to which one wants to navigate).

  This is natural, since the role name and the multiplicity on the opposite end of the assocation determine the pointer attribute for this class.

Collection-Type (1)

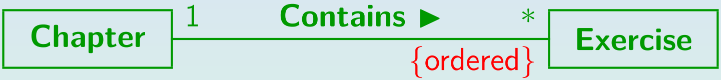
- Consider again the relationship between chapters and exercises:



- As explained above, in an OODB, the class “Chapter” will contain a set-valued attribute with pointers to “Exercise” objects.
- If one iterates over the elements of this set, they are returned in no specific order.

Collection-Type (2)

- However, one can specify in UML that the order of exercises in a chapter is significant:



- Then not a set, but a list will be used to hold the pointers to exercises (but duplicates are still not allowed).
- "{ordered}" can also be used on one or both sides of a many-to-many relationship.

  Only for multiplicities 0..1 and 1 it makes no sense.

Collection-Type (3)

- In the ODMG proposal, **set**, **list**, or **bag** can be used in relationships:

```
class Chapter (extent chapters)
{   ...;
    relationship list<Exercise> contains
                      inverse Exercise::belongs_to;
};
```

- However, one must exclude duplicates from the list.

  An “ordered set” as in UML is not quite the same as a list.

Collection-Type (4)

- In a relational implementation, one would add a number to the exercises table (exercise number within chapter) in addition to a foreign key referencing the chapter.

  `EXERCISES(ID, ..., CHAPTER→CHAPTERS, SORT_NO)`

- `CHAPTER` and `SORT_NO` together are an alternative key for `EXERCISES`.

  This ensures that there is really a defined sequence for the exercises within one chapter.

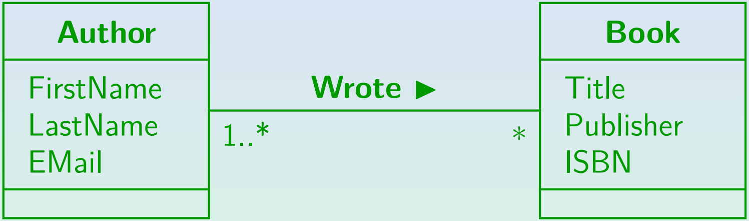
Collection-Type (5)

- Note that "{ordered}" means that additional information needs to be stored besides the set of links between objects.
- If the exercise objects already contain an exercise number, so that the order can be derived from this information, "{ordered}" would not be correct (redundant information).

  One can use "{sorted}" to indicate that for a more efficient implementation, it would be good to store the links sorted by some criterion, e.g. the exercise number.

Exercise

- Consider the following class diagram:



- If a book has several authors, their sequence is important (it is not always the alphabetical sequence). How would you specify that?
- Translate this diagram into the relational model.

References

- Grady Booch, James Rumbaugh, Ivar Jacobson:
  The Unified Modeling Language User Guide.
  Addison Wesley Longman, 1999, ISBN 0-201-57168-4, 482 pages.
- James Rumbaugh, Ivar Jacobson, Grady Booch:
  The Unified Modeling Language Reference Manual.
  Addison Wesley Longman, 1999, ISBN 0-201-30998-X, 550 pages, CD-ROM.
- Martin Fowler, Kendall Scott: UML Distilled, Second Edition.
  Addison-Wesley, 2000, ISBN 0-201-65783-X, 185 pages.
- Terry Quatrani: Visual Modeling with Rational Rose 2000 and UML.
  Addison-Wesley, 2000, ISBN 0-201-69961-3, 256 pages.
- Robert J. Muller: Database Design for Smarties — Using UML for Data Modeling.
  Morgan Kaufmann, 1999, ISBN 1-55860-515-0, ca. $40.
- Paul Dorsey, Joseph R. Hudicka: Oracle8 Design Using UML Object Modeling.
  ORACLE Press, 1998, ISBN 0-07-882474-5, 496 pages, ca. $40.
- OMG's UML page: [http://www.omg.org/technology/uml/index.htm]
- UML 1.3 Specification: [https://www.omg.org/spec/UML]
- UML Resources: [https://www.uml.org/resource-hub.htm]