Datenbanken II A: DB-Entwurf

Chapter 12: UML Class Diagrams II

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:

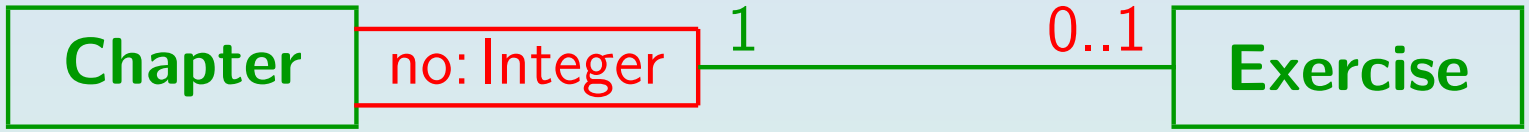
- read and write UML class diagrams.
- translate ER-schemas into UML class diagrams and vice versa.
- translate a UML class diagram into a relational database schema (as far as possible).
- explain differences between the object-oriented and the classical relational approach to database design.

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

Contents

Qualifiers (1)

- If each exercise has a unique number within a chapter, this can be expressed by means of a “qualifier”:



- Chapter objects now basically contain an array of links to Exercise objects.
- The array is indexed by a number, and returns 0 or 1 exercises for a given number.

  A normal association would map “Chapter” objects into sets of “Exercise” objects. Now a “Chapter” object and a value for the qualifier “no” are mapped into at most one “Exercise” object.

Qualifiers (2)

- More general, the qualifier can be of any data type, e.g. also a string. Then a "dictionary" data structure would be stored within the Chapter objects, e.g. a hash table or a search tree.
- Arrays and dictionaries are also collection types.

  Qualifiers are strongly related to the use of "{ordered}" etc. to determine the collection type of the association. An array could be used to implement an ordered association (at least if the maximum number of related objects is known), but the qualifier makes clear that the specific value of the array index is important for the application.

Qualifiers (3)

- A qualifier can also be used when there is more than one related object for a given qualifier value.

  I.e. the qualifier only partitions the set of related objects into subsets. It could then be represented by an attribute in an association class (or in the target class), but the qualifier makes clear that some kind of efficient access should be possible.

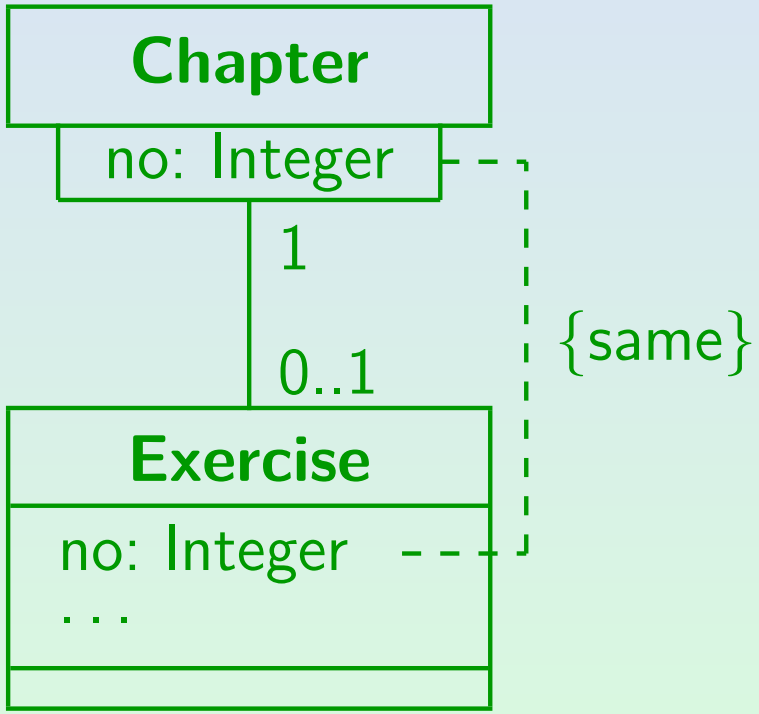
- The multiplicity on the opposite association end is influenced by the qualifier: E.g. `0..1` at the Exercise end is the number of objects that may be related to a single Chapter object for a given qualifier value.

  So "Chapter" and the qualifier now form some kind of composite object for the purpose of determining multiplicities.

Qualifiers (4)

- With qualifiers, UML gets something like keys, but only in the context of a given object.
- The situation is similar to a weak entity, but the qualifier value (the exercise number) is not part of the Exercise class.
- If that is required, the exercise number must be stored redundantly as an attribute of the Exercise class, and a constraint is needed to enforce the equality (see next slide).

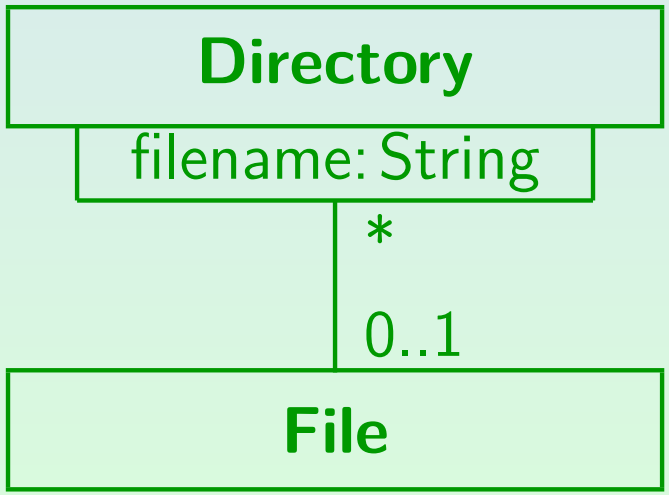
Qualifiers (5)



Qualifiers (6)

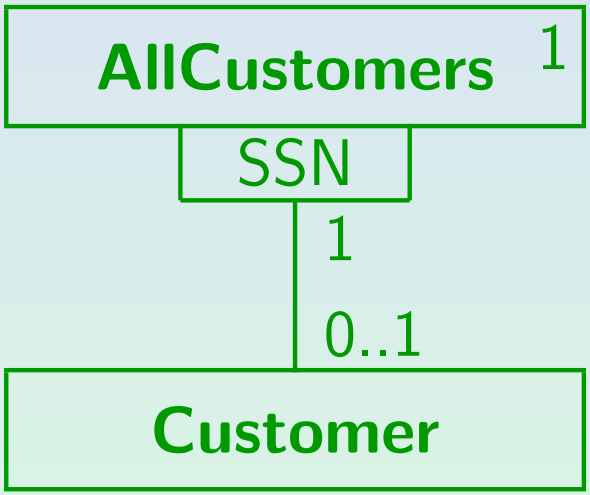
- In the UNIX file system, the filename is not part of the file objects, but appears only in the directory. This would be a classical example of a qualifier.

  The same file may actually appear in different places of the file system (in different directories or under different names).



Qualifiers (7)

- If one has a globally unique object a qualifier from there corresponds to a key:



If direct access from a customer to his/her social security number is needed, a duplication of SSN as shown on slide 8 is required.

- Does it have to be so complicated?

Qualifiers (8)

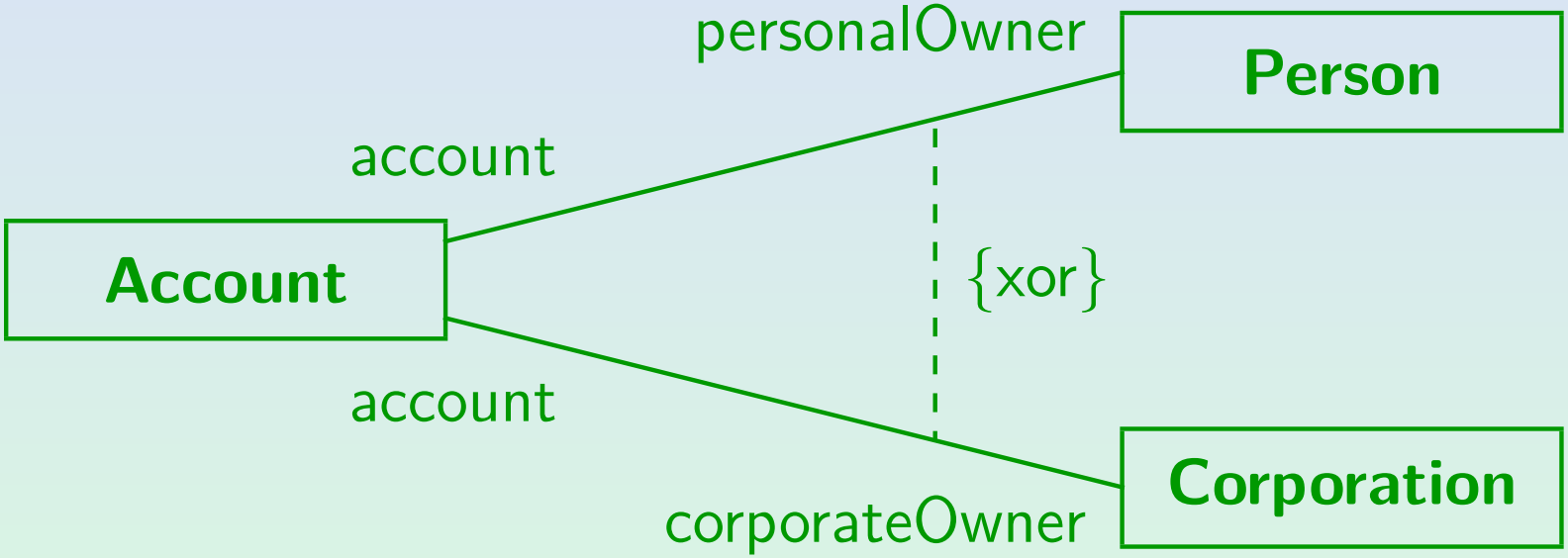
- The "AllCustomers" object is in effect a unique index that supports the key "SSN" for the class "Customer".
- If UML is not extended in order to support keys, one must show the index explicitly as in this example.
- This is clearly a relapse to pre-relational times.

  An index is something big and complicated, so one might argue that when designing an object-oriented program (e.g., in C++), the index should be shown explicitly if it is needed. However, when designing a database, creating an index is easy, and furthermore indexes should not be part of the conceptual design. By the way, the ODMG model has the notion of keys (for extents).

Contents

Constraints on Associations (1)

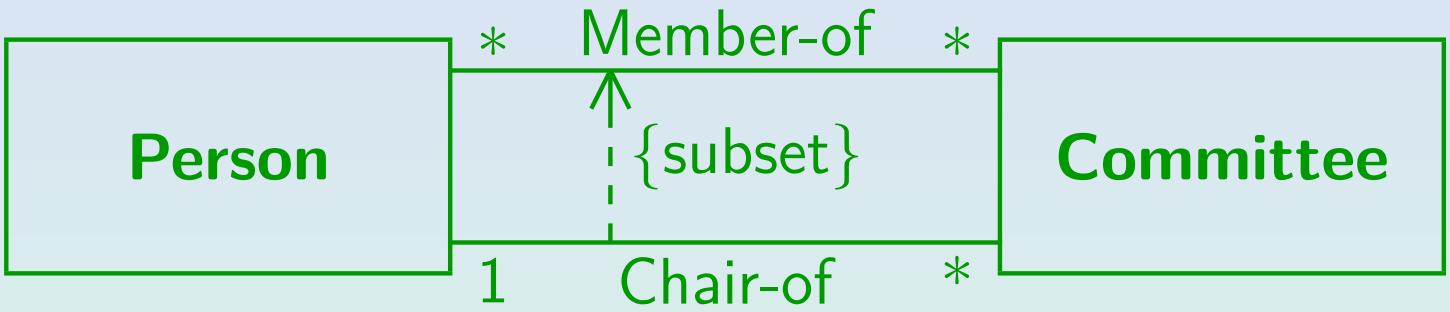
- One can specify that two associations exclude each other:



[Rumbaugh et.al.: The UML Reference Manual, 1999, p. 156]
Note that "xor" is not quite right: If the minimum cardinality is 0, it is possible that an Account has no link at all.

Constraints on Associations (2)

- One can specify that an association implies another one:



[Rumbaugh et.al.: The UML Reference Manual, 1999, p. 237]

- As attributes, associations can be marked
  - `changeable` (the default),
  - `addOnly` (links can only be inserted, not deleted)
  - `frozen` (links of an object cannot be changed).

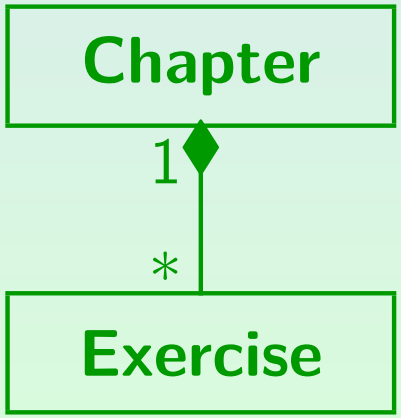
Contents

Composition/Aggregation (1)

- Composite aggregation (or composition) is the relationship between a whole and its parts.

  Or really vice versa: parts are aggregated to a whole.

- An association becomes a composition (a form of aggregation association) by marking the side of the whole with a black diamond:



Composition/Aggregation (2)

- An object can only be part of one composite object at a time:
  - The multiplicity on the side of the composition must be `1` or `0..1`.
  - From every class, there can be at most one outgoing composite aggregation relationship.

    Actually, there could be more, but they must be linked with a `xor`-constraint.
- On the instance level, composite aggregations may not be cyclic (an object cannot be part of itself).
- On the class level, recursive composition relationships are allowed: A class has many objects, so an object of a class may be part of another object of the same class.

Composition/Aggregation (3)

- The whole is responsible for disposing its parts:
  If the whole is deleted, it must delete its parts.
  - In relational databases, this means that the foreign key is specified with `ON DELETE CASCADE`.
  - In C++, the destructor for the composite object would call the destructors for its parts.

    In C++, there is no automatic garbage collection, so one needs to think about memory management.
- It is legal that
  - a part is created after the composite or destroyed before it,
  - a part is moved from one composite object to another,

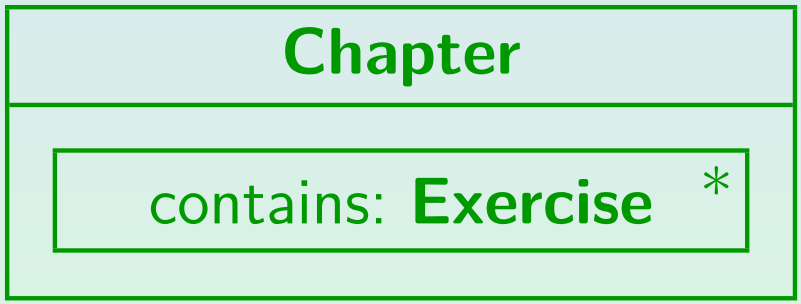
  but this would normally be done by operations of the composite object (it manages its parts).

Composition/Aggregation (4)

- Alternative notation: The part class is drawn within the rectangle for the composite class.

  Composition is the relationship between an object and its attributes.

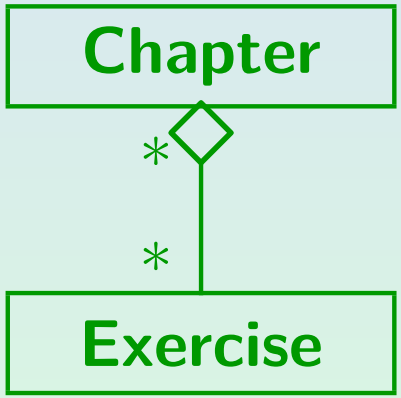
  Attribute name: role name of the part.



- If an association is drawn within the boundaries of the rectangle of the composite class, it can exist only between parts of the same composite object.

Composition/Aggregation (5)

- UML also has a weak form of aggregation, called “simple aggregation” or “aggregation”.
- It is denoted by an open diamond:



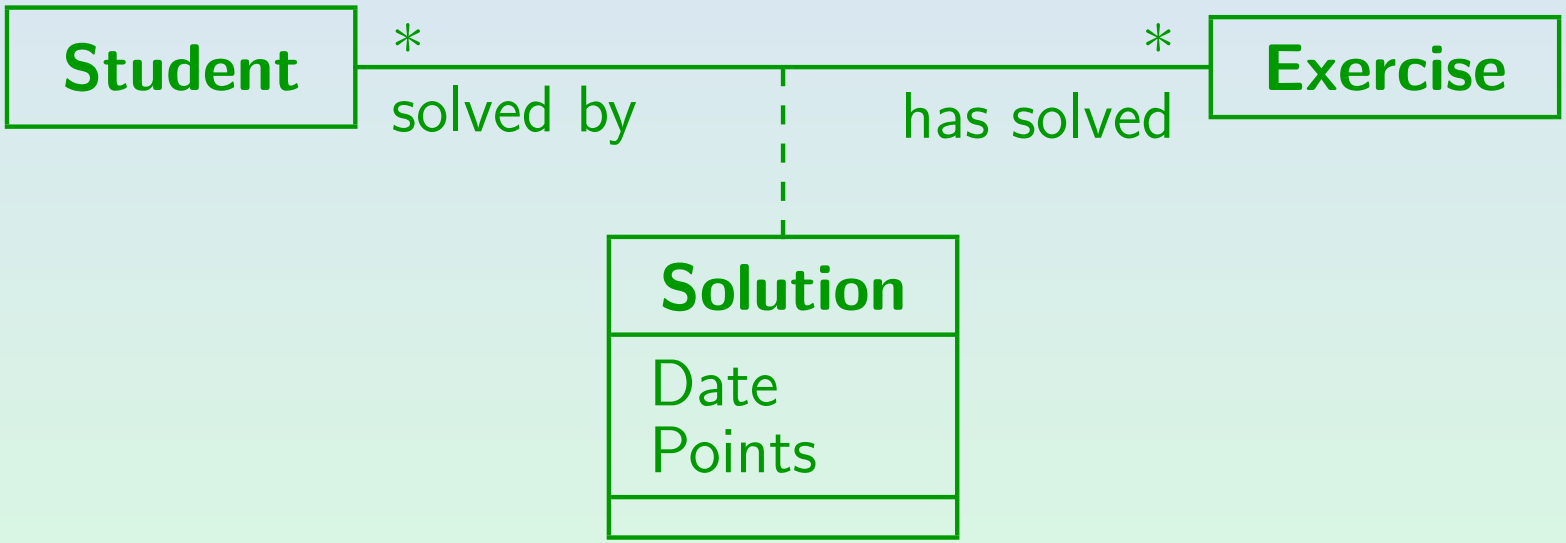
- It has no semantic consequences: An object can be part of more than one aggregated object.

“Think of it as a modeling placebo” [Rumbaugh cited after Fowler, 1999].

Contents

Association Classes (1)

- If an association has attributes (or operations), an "association class" must be used:



- An association class is shown as a class that is linked by a dashed line to an association.

Association Classes (2)

- There is exactly one object of the association class "Solution" for every pair of objects from Student and Exercise that are linked via the association.
- In UML, there cannot be two links between the same two objects via the same association.

  I.e. associations are sets (as relationships in the ER-model).
- Thus, the above class diagram enforces that the same student cannot submit two solutions for the same exercise.

Association Classes (3)

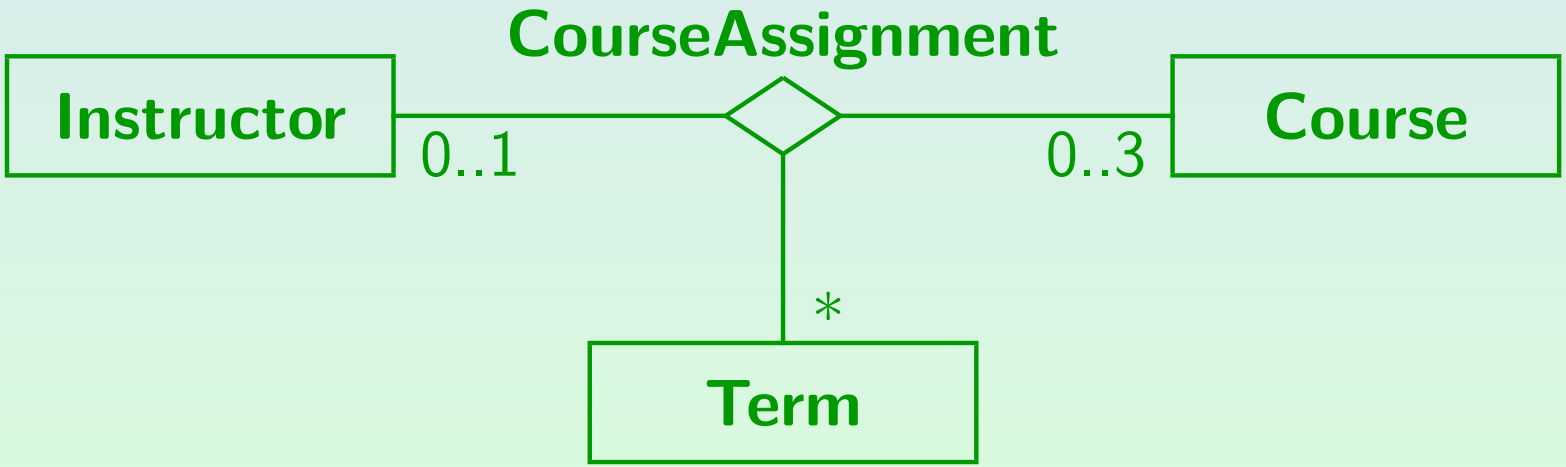
- In this schema, the same student can have two (or more) solutions for the same exercise:

In the ER-model, "Solution" would be a weak entity with owners "Student" and "Exercise". Then the constructed key enforces the required uniqueness. But in UML, one can specify keys only via user-defined extensions to the standard UML syntax.

- For one-to-many associations, attributes of the association can be added to the class at the "many" side. An association class is not required.

Non-Binary Associations (1)

- UML is not restricted to binary associations, although that is by far the most common case.
- An *n*-ary association is symbolized by a diamond with *n* connections to the participating classes:



Non-Binary Associations (2)

- The multiplicities specify how many objects of that class can exist for a given combination of objects from the other classes.

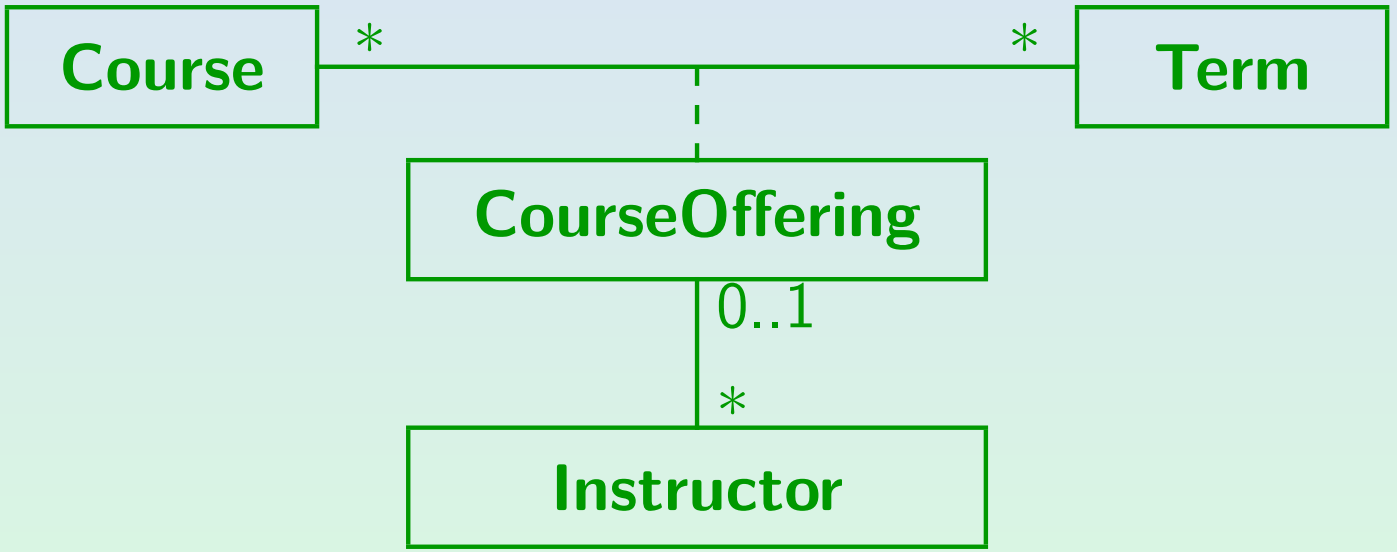
  E.g. the same instructor can offer in the same term not more than three courses. For a given course and a given term, there is at most one instructor. Zero instructors would mean that this combination of course and term do not appear in the association. With this ternary association, it is not possible to store that a course is offered in a term, but with a yet unknown instructor.

- Navigability, aggregation, and qualifiers are not permitted for non-binary associations.

  Their semantics is too complicated.

Non-Binary Associations (3)

- If there is only one instructor per term for a course, the following model might be better:



This permits to store course offerings for which an instructor is not yet assigned. It does not permit multiple sessions of the same course in the same term. It does not enforce the maximal teaching load.

Contents

Operations (1)

- "An operation is a specification of a transformation or query that an object may be called to execute. It has a name and a list of parameters."
- "A method is a procedure that implements an operation. It has an algorithm or procedure description."
  [Rumbaugh et.al.: The UML Reference Manual, 1999, p. 369.]
- UML distinguishes between
  - operations (the interface) and
  - methods (the implementation).

Operations (2)

- E.g. if an operation *o* from the superclass is overridden in the subclass, there is one operation and two methods.

  Most people do not take this distinction very strictly.

- "An operation is the implementation of a service that can be requested from any object of the class to affect behavior. In other words, an operation is an abstraction of something you can do to an object and that is shared by all objects of that class."

  [Booch et.al.: The UML User Guide, 1999, p. 51.]

Operations (3)

- UML allows that there are two operations with the same name, but different lists of parameter types.

  This corresponds to the overloading of functions in languages like C++: The compiler can decide by the types of the arguments in the function call which function is meant.

- UML is used to specify programs written in C++, Java etc. Thus, the basic C++ constructs should be expressible in UML.

Operations (4)

- A full operation declaration consists of:
  - Visibility: + (public), # (protected), – (private).

    The visibility specification is optional.
  - The name of the operation.
  - The parameter list, enclosed in "(" and ")".

    The parameters can be suppressed. But even if only the name is shown, it is usually followed by () to make clear that this is an operation and not an attribute.
  - A colon and the return type.

    This is optional. The default is "null" (i.e. no result).
    In UML even a list of return types is possible.
    Parameter list and return type can only be suppressed together.

Operations (5)

- The parameter list is a comma-separated list of parameter declarations consisting of
  - A direction (optional): `in`, `out`, or `inout`.

    The default is `in` (input parameter, i.e. read-only access).
  - Parameter name, colon "`:`", and parameter type.
  - An equals sign "`=`" and a default value for the parameter.

    This is optional. If a default value is declared, a call to the operation does not have to specify a value for the parameter. This is also a feature of C++: E.g. if a function has two parameters, but a default value for the second one is declared, it can be called with one parameter.

Operations (6)

- Example of an operation declaration:

```
+getTotal(StudID: Integer,
          InclExtra: Boolean = true): Float
```

- In front of an operation declaration, a stereotype can be specified. It is enclosed in «...».

  A stereotype can even apply to an entire group of operations. In a list compartment (e.g. attributes, operations), stereotypes can be specified as list elements by themselves. Then they apply to all following list entries until the next stereotype that appears as a list element.

- After an operation declaration, a property list can be specified. It is enclosed in {...}.

Operations (7)

- The scope of an operation can be “instance” or “class”. Operations of class scope are marked by underlining.
  - Operations of instance scope apply to individual objects, so they have a hidden parameter for an object of their class.
  - Operations of class scope apply to the class as a whole, not a specific object. Therefore, they can access only attributes of class scope.

Operations (8)

- An operation may be declared a query operation (stereotype keyword «query»). Then this operation is guaranteed not to modify the state of the object.

  It is equivalent to specify the property `isQuery=true`. The default is `isQuery=false`, i.e. the operation can assign values to the attributes and change associations.

- Operations can be marked as «constructor».
  Such operations create and initialize instances (objects) of the class.

  They have class scope, but can access the attributes of the newly created instance. They implicitly return the created instance, but no return type needs to be specified.

Hiding Attributes (1)

- The main difference between the object-oriented and the relational approach are the operations.

  Of course, generalization and non-atomic attributes are nice object-oriented features, which relational databases would like to have (this lead to object-relational DBs). But the cultural clash lies in operations and identity.

- Usually, all attributes are declared as private and can only be accessed via operations of the class.

  Of course, one can have public attributes in UML and e.g. in C++, but this is generally considered bad style. E.g. in Smalltalk-80, it was impossible: "A crucial property of an object is that its private memory can be manipulated only by its own operations." [Goldberg/Robson, 1983, p. 6]

- Classes often have operations `get_A` and `set_A` for many of their attributes `A`.

  This is especially true if the class basically corresponds to a relation.

Hiding Attributes (2)

- The reason why the object-oriented approach distinguishes between private attributes and public operations is that
  - the implementation can be changed
  - while the interface is kept stable.
- In relational databases, this corresponds to physical data independence: E.g. indexes can be changed while the table structure remains stable.
- In relational databases, the table structure normally is the interface, it does not need to be hidden (except for security purposes, but that is a different issue).

  In the ANSI/SPARC architecture, there is a second interface level that gives logical data independence.

Hiding Attributes (3)

- Complex programs like compilers or DB management systems have a relatively small user interface, but difficult algorithms. Different levels of interfaces (system layers) are needed.
- DB application system have a large user interface (many screens), but simple algorithms. Thus, a single level distinction between interface and implementation might be enough.

Hiding Attributes (4)

- Basically, somebody who invested money and work to build a relational database does not understand why he/she should restrict the access to the data by permitting only to call query operations, not direct read access to all attributes.

  Having to write program code for queries is a step back from the declarative language SQL.

- Views usually only extend the interface, but do seldom hide details below them (except for security).

Implementing Operations (1)

- Of course, query operations that are not simply a "get attribute", but compute some derived value, are interesting for relational databases, too.
- They can normally be mapped into view definitions.
- In order to avoid unnecessary joins, one will often have one view for a relation that gives access to all explicitly stored attributes as well as all derived attributes (query operations).

  If, however, a join is necessary for the computation of the result of the query operation, it might be better to have it in a distinct view.

Implementing Operations (2)

- Query operations with parameters are not in general implementable in this way.

  If the parameter can take only values that appear in the database (or else one of a few enumeration constants), the parameter can be implemented as an attribute of the view. Otherwise, this method does not work since views must be finite. (Deductive DBs have "binding restrictions" for this purpose, i.e. values for certain attributes must be specified.)

- If necessary, operations can be mapped to stored procedures or procedures in a library for developing application programs.

  This is also necessary if the algorithm cannot be expressed in SQL, e.g. requires a transitive closure.

Implementing Operations (3)

- For attributes that participate in complex constraints, it is useful to exclude direct write access via `UPDATE`, and permit changes only via procedures (operations of the class).
- Some other attributes should be non-updateable (E.g. attributes participating in a primary key.)
- So for write accesses, the object-oriented distinction between the internal state (attributes) and the external interface (operations) might make sense.

Implementing Operations (4)

- The more data structure invariants need to be protected, the more important it is to exclude direct attribute modifications.
- Direct updates can be excluded if
  - the tables are installed under an account that is only used by the DBA,
  - real users (and programs) log in under a different account and can be granted selective access rights.

    Especially, they get update rights only for certain attributes.

Implementing Operations (5)

- Operations can be implemented as stored procedures on the server, or library procedures that are linked to client programs.

  Library procedures don't give the access protection.

- Triggers can be used if the operation mainly sets an attribute, but additional constraints need to be checked and redundantly stored values (e.g. sums) must be updated.

Contents

Generalization (1)

- "A generalization is a relationship between a general thing (called the superclass or parent) and a more specific kind of thing (called the subclass or child). Generalization is sometimes called an "is-a-kind-of" relationship."

  [Booch et.al.: The UML User Guide, 1999, page 64/141]

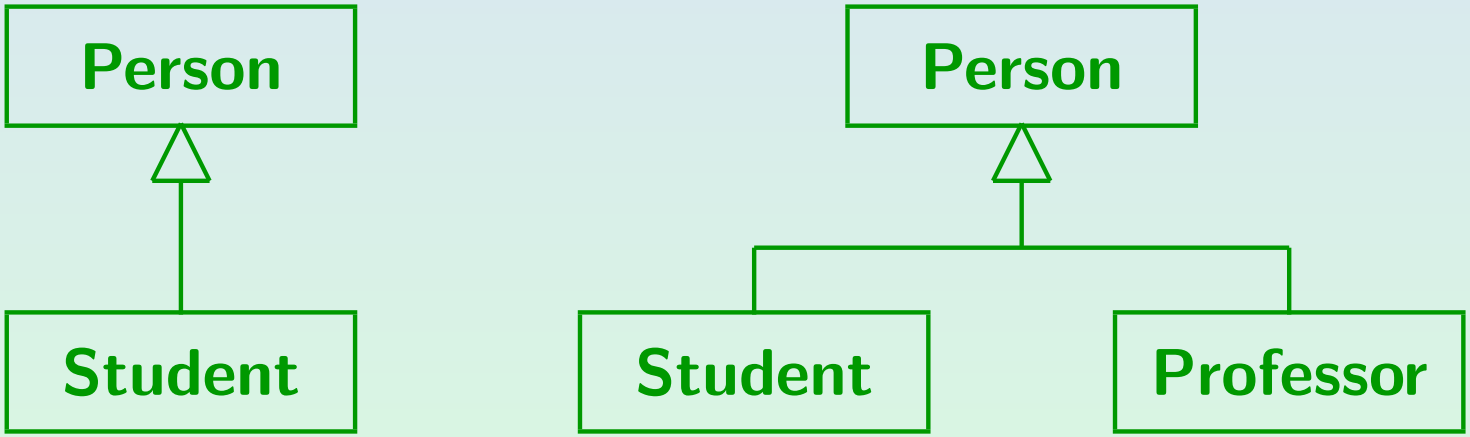
  Generalization: "A taxonomic relationship between a more general element and a more specific element. The more specific element is fully consistent with the more general element and contains additional information."

  [Rumbaugh et.al.: UML Reference Man., 1999, p. 287]

- The four kinds of relationships in UML are: Dependency, Association, Generalization, Realization.

Generalization (2)

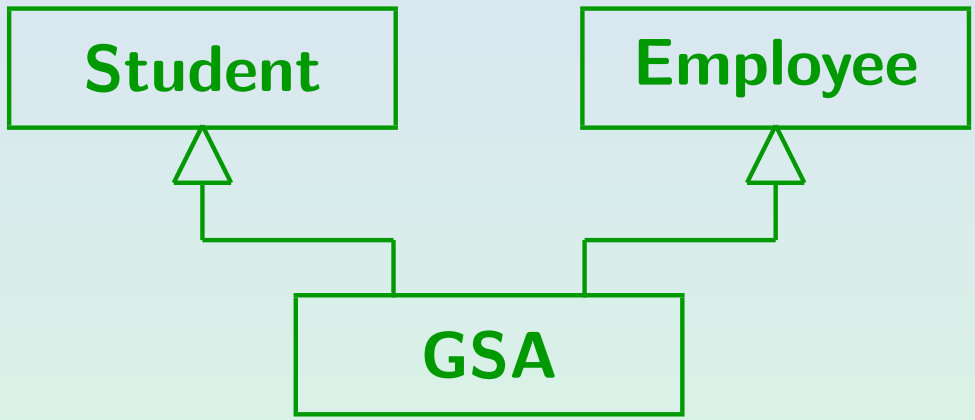
- Generalization is shown in UML as an arrow (with a large open triangle at the end) pointing from the subclass to the superclass (in the "is a" direction):



- If a class has several subclasses, either single arrows can be used or the combined "tree notation".

Generalization (3)

- Multiple inheritance is allowed in UML, i.e. a class can have two or more superclasses:



- "Use multiple inheritance carefully. You'll run into problems if a child has multiple parents whose structure or behaviour overlap."

  [Booch et.al.: The UML User Guide, 1999, p. 142.]

Generalization (4)

- Of course, the subclass can be a superclass for other classes, i.e. there can be a whole hierarchy of subclass-superclass relationships.

  Cycles are forbidden. Generalization is a transitive, anti-symmetric relationship (partial order, lattice). So transitive edges (directly to a super-super-class) should semantically change nothing. In practice, they should be avoided.

- The superclass is also called parent of the subclass, direct and indirect superclasses its ancestors. Correspondingly, the subclass is called child of the superclass, direct and indirect subclasses its decendants.

Inheritance (1)

- A Subclass inherits structure and behaviour, i.e. attributes and operations, from its superclass.
- An instance of the subclass can be used in any context where an instance of the superclass is required.

  The value of a variable/parameter of type S can actually be an instance of a subclass of S. Liskov substitutability principle.

- If the generalization arrow is marked with the stereotype «implementation», the inherited attributes and operations become private.

  This is not a real use of generalization, since the basic substitutability principle is violated. C++ has such a notion of "private inheritance".

Inheritance (2)

- In the model/diagrams, only attributes and operations are shown that are added to the inherited ones.

  It is illegal in UML to redeclare an inherited attribute. An inherited operation may be redeclared to show overriding.

- In case of multiple inheritance, it is forbidden if a class inherits the same attribute/operation from two different classes.

  Then it would be unclear which of the two methods for the operation should be used. Of course, it is legal if the operation is inherited from a common superclass on two different inheritance paths.

- To override an inherited operation (usually) means to replace its implementation (method) for objects of the subclass.

  However, complicated techniques for combining the inherited method with method declared in the subclass have been proposed and UML does not require the simple replacement semantics (depends on programming language).

Inheritance (3)

- Operations have a property `isPolymorphic`.
  If it is `false`, the operation cannot be overridden.

  The default value is `true`. In C++, polymorphic operations must be declared as `virtual` (called via a pointer in the object: "late binding").

- A class can have the property `leaf`, in which case it is not legal to declare a subclass of it.

  In the same way, there is a property `root` which means that this class cannot have a superclass. Operations can also be declared as `root` or `leaf`, `leaf` seems to mean the same as `isPolymorphic=false`. A polymorphic operation may be declared `leaf` in a descendant class which means that further down in the hierarchy it cannot be overridden.

Abstract Classes (1)

- An abstract class is a class that cannot have direct instances, i.e. there can be no objects of this class.
- However, subclasses of the abstract class can have instances.

  Otherwise, the class could only be interesting because of operations of class scope.
- Abstract classes correspond to total specialization.
- Abstract classes / abstract operations (see below) are marked by writing their declaration in italics.

Abstract Classes (2)

- Abstract classes can have abstract and concrete operations:
  - For a concrete operation, a method (implementation) is already specified in the abstract class.
  - For an abstract operation, a method must be specified in each subclass.

    Abstract operations must be polymorphic since they can only be used when the non-existant implementation is overridden (in C++: pure virtual functions).

Generalization Constraints (1)

- A generalization can be marked as "{complete}" which means that all possible subclasses have been declared and no further subclasses may be added.

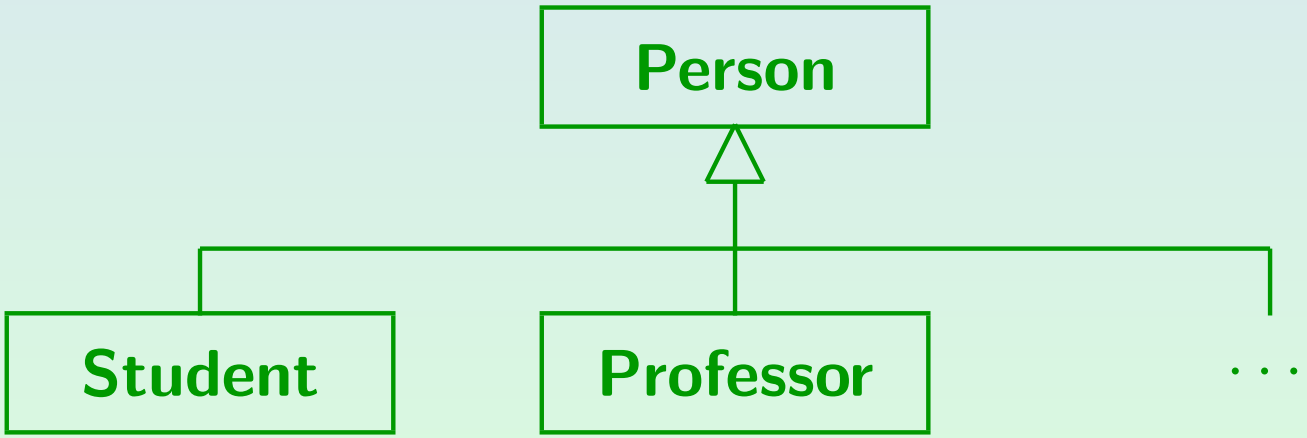
  A generalization can be marked as complete even if not all subclasses are shown on the diagram. It suffices that all have been declared in the model.

- Conversely, it can be marked as "{incomplete}" which means that more subclasses are known or expected but have not been declared yet.

  Note that this is not the same as total and partial specialization in the ER-model. E.g. the UML Reference contains incomplete generalization with an abstract superclass (p. 290).

Generalization Constraints (2)

- It is possible to use an ellipses symbol in a diagram to mark that there are more subclasses that are not shown on the diagram ("elided"):



Generalization Constraints (3)

- A generalization can be marked as "{disjoint}" or "{overlapping}".
- Disjoint means that an object of the superclass can only have one of the subclasses as type.
- E.g., if "Person" has subclasses "Student" and "Employee", and both are declared {disjoint}, it is impossible to later introduce a class "GSA" that has both, Student and Employee, as superclasses.

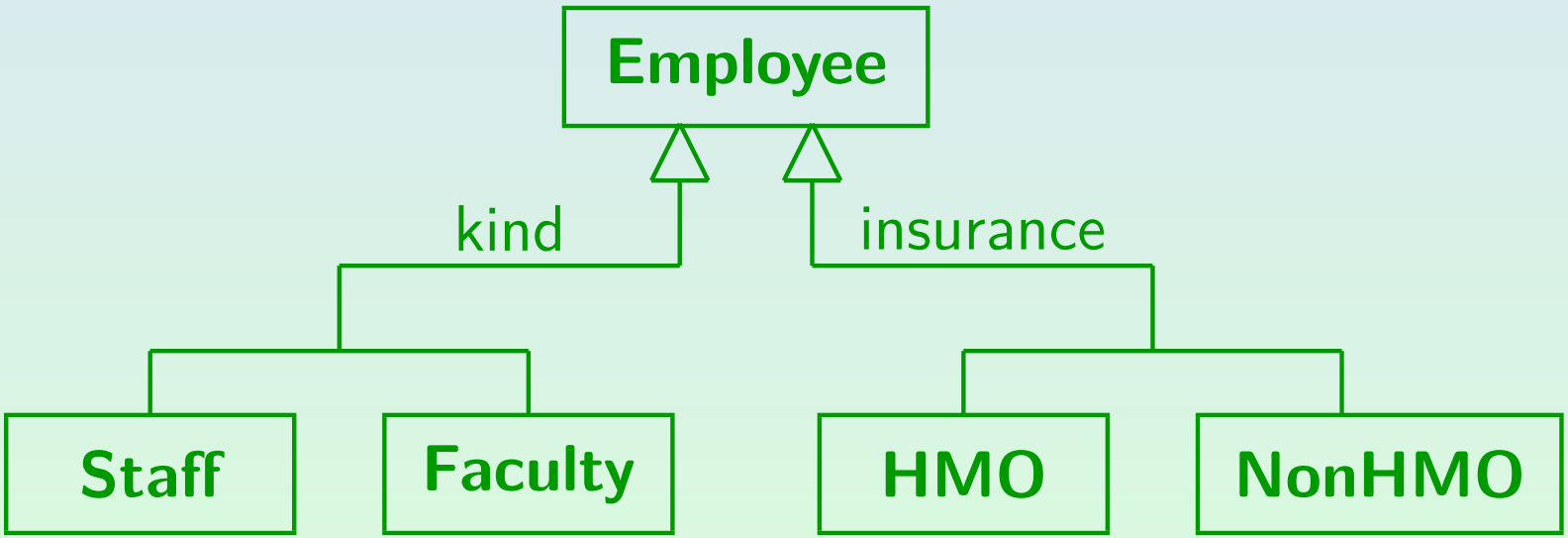
Multiple Classification (1)

- In most programming languages, objects must have a unique "direct class" (i.e. most specific class).
- It is then automatically an indirect instance of all ancestors (superclasses etc.) of its direct class.
- UML permits "multiple classification", i.e. an object can be a direct instance of more than one class.

  This basically corresponds to multiple inheritance with anonymous subclasses. E.g. with multiple classification, an object can be at the same time "Student" and "Employee", even if no "GSA" class is explicitly declared. If there are a lot of possible combinations, it would be too much effort to declare them all explicitly.

Multiple Classification (2)

- Generalization arrows can be marked with “discriminators” (names) to show the different dimensions along which objects can be classified:



Multiple Classification (3)

- "All subtypes with the same discriminator are disjoint; that is, any instance of the supertype may be an instance of only one of the subtypes within that discriminator." [UML Distilled, 2nd Ed, 2000, p. 83]
- "A parent with multiple discriminators has multiple dimensions, all of which must be specialized to produce a concrete element. Therefore, children within a discriminator group are inherently abstract. [...] A concrete element requires specializing all the dimensions simultaneously." [UML Ref. Man., p. 262/263]
- If no discriminators are specified, all generalizations with the same parent form one discriminator group. (Consistent?)
- Discriminators become attributes of the instances.

Dynamic Classification

- Dynamic classification means that an object can change its class over time.

  Most programming languages use static classification: The type of an object is fixed at runtime.

- This is normally used together with multiple classification: An object has a static base class and can gain or lose additional "roles" over time.

  Fowler uses the stereotype ⟨⟨`dynamic`⟩⟩ on the generalization relationship. It does not appear in the UML Reference or the User Guide.

  The Reference Manual says dynamic or static classification is a semantic variantion point and that either assumption may be used in a UML 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]