CS 332/780: Object-Oriented Databases Spring 2018
Course Notes #4
Keitaro Yukawa
Department of Computer Science
Queens College, CUNY

# Modeling Relations in UML

*N*-ary relations, $N \geq 2$, are used to model relationships among class objects. In the following we will often regard a class name as denoting the extent of the class, i.e., the set of objects currently existing in the class, and use the membership notation $x \in C$ to mean x is an object of class C.

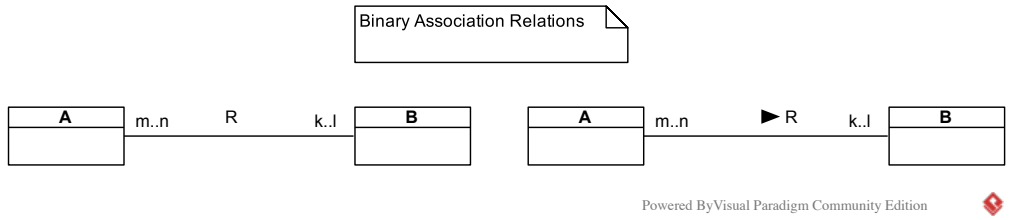
## 1 Binary Relations

When a binary relation R(A, B) is established between classes A and B in UML database models, it is normally understood that R can be traversed in both directions. This means that for any given $a \in A$, $R{\downarrow}B(a) = \{ b \in B \mid R(a, b) \}$ is retrievable, and for any given $b \in B$, $R{\downarrow}A(b) = \{ a \in A \mid R(a, b) \}$ is retrievable. In some cases R's direction from A to B may imply that traversals from A to B are more frequent and/or efficient than from B to A. Other than this, UML models of binary relations specify nothing about the nature of their implementations. (UML does provide a means of restricting traversals to one direction only – this issue will not be covered in this course.) The relation R is inherited to the descendant classes of A and B in the sense that R(x, y) may hold between objects x and y in any descendants of A and B, respectively.

Two types of binary relations are used in UML database modeling: *association* relations and *whole-part* relations.

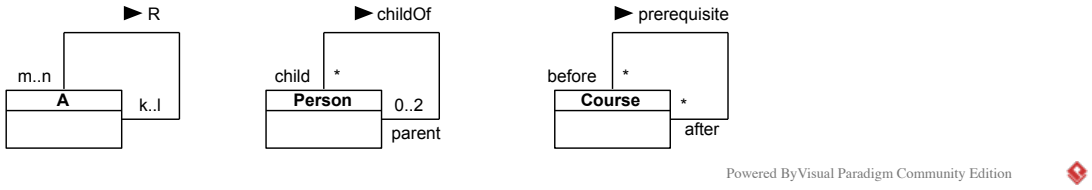
### 1.1 Association Relations

An association relation is a binary relation R(A, B) between classes A and B where neither A-objects nor B-objects are "parts" or "components" of the others, conceptually, organizationally, or physically. Graphically, it is represented by a line between A and B labeled with the relation name R. Optionally, the direction of the relation from A to B may be indicated by attaching a small black triangle to the relation name, or by putting an arrowhead at the B end of the line.




On each end of the line is shown the range of multiplicities using the notation x..y where x, y are non-negative integers with $x \leq y$. The range k..l on B's end requires that the condition $k \leq |R{\downarrow}B(a)| \leq l$ hold for each $a \in A$ at all times, and m..n on A's end requires that $m \leq |R{\downarrow}A(b)| \leq n$ hold for each $b \in B$ at all times. If the upper bound y of x..y is unknown or unspecified, use $*$, which stands for an arbitrary non-negative integer. For example, the range $1..*$ means "at least one but no upper bound". The range $0..*$ means neither upper nor lower bound is imposed, and may be abbreviated as $*$. A range n..n may be abbreviated as n. Non-contiguous unions of ranges may also be used, such as "1..5, 10..20", "1..10, 20..*". The lower and upper bounds of multiplicities should be determined by the semantics of relation R in the context of the application domain.

Binary association relations R(A, A) from A to itself are represented by a line from A to itself. Since the source and target classes are the same, it is helpful to indicate the relation's direction explicitly and to attach *role names* at both ends of the relation.




The 2nd diagram represents the relation childOf(Person, Person) where $\text{childOf}(p_1, p_2)$ means $p_1$ is a child of $p_2$. For each person $p_1$, $\text{childOf}{\downarrow}\text{Person}_2(p_1) = \{ p_2 \mid \text{childOf}(p_1, p_2) \}$ is the set of $p_1$'s parents and is required to have 0, 1, or 2 persons, as one or both of the parents may not be known and/or not entered into the database. For each person $p_2$, $\text{childOf}{\downarrow}\text{Person}_1(p_2) = \{ p_1 \mid \text{childOf}(p_1, p_2) \}$ is the set of $p_2$'s children and may have any number of persons. $\text{childOf}{\downarrow}\text{Person}_1 = \{ p_1 \mid \exists p_2\ \text{childOf}(p_1, p_2) \}$ is the set of persons who have at least one parent object in the database, and $\text{childOf}{\downarrow}\text{Person}_2 = \{ p_2 \mid \exists p_1\ \text{childOf}(p_1, p_2) \}$ is the set of persons who have at least one child object in the database.

### 1.2 Whole-Part Relations

A whole-part relation is a binary relation R(A, B) between a whole class A and a part class B where each B-object is, or may be, a part or component of some A-object functioning as a whole, conceptually, organizationally, or physically. Every A-object is semantically understood as a whole entity that includes/contains B-objects. As to determination of what sorts of things are to be modeled as parts, it is adequate to exercise case-by-case, prudent applications of dictionary meaning of *part*. The following is an excerpt from Merriam-Webster Dictionary:

1. one of the often indefinite or unequal subdivisions into which something is or is regarded as divided and which together constitute the whole; an essential portion or integral element
2. one of several or many equal units of which something is composed or into which it is divisible
3. a constituent member of a machine or other apparatus
4. something falling to one in a division or apportionment; share
5. one's share or allotted task (as in an action); duty
6. a general area of indefinite boundaries
7. a function or course of action performed
   **synonyms**: portion, piece, member, division, section, segment, fragment

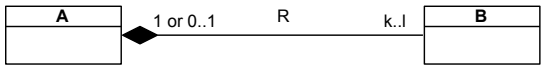
Two types of whole-part relations are distinguished: *composition* relations and *aggregation* relations.

**Composition Relations (Strong Composition Relations)** When a whole-part relation R(A, B) is composition, the following constraints are imposed on the relation:

- **Uniqueness of the whole**. $|R{\downarrow}A(b)| \leq 1$ for each $b \in$ B at all times, that is, each B-object may be a part of at most one A-object at any time. No B-object is ever shared by multiple A-objects.
- **Parts' lifetime dependence on the whole**.
  - It is not permitted that R(*a*, *b*) and R(*a'*, *b*) hold for any distinct whole objects *a*, *a'* and for any part object *b* at any distinct times, that is, no part object is permitted to change the whole object.
  - When R(*a*, *b*) holds for any whole object *a* and any part object *b* and *a* is deleted from the class A, *b* too must be deleted from the class B.

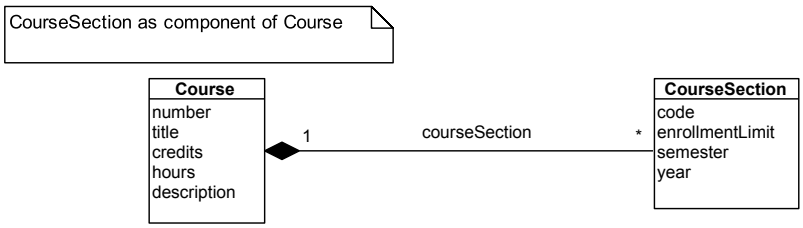
The last condition implies that whenever there is a chain of composite relations, deletion of a whole object will cause a cascade of deletions, i.e., deletion of its parts, sub-parts, sub-sub-parts, and so on, which are all related by the composite relations. The underlying intuition is that as long as an object is a part of a whole, it will "live and die" with the whole. Note however that a B-object may exist without being a part of any A-objects.

A composition relation is graphically represented by a line between A and B labeled with the relation name R, with a small black diamond at A's end. The range of multiplicities on both sides should be shown using the same x..y notation used for association relations. The upper bound on the multiplicity for A's side is always 1 as each B-object can be a part of at most one A-object at any time.



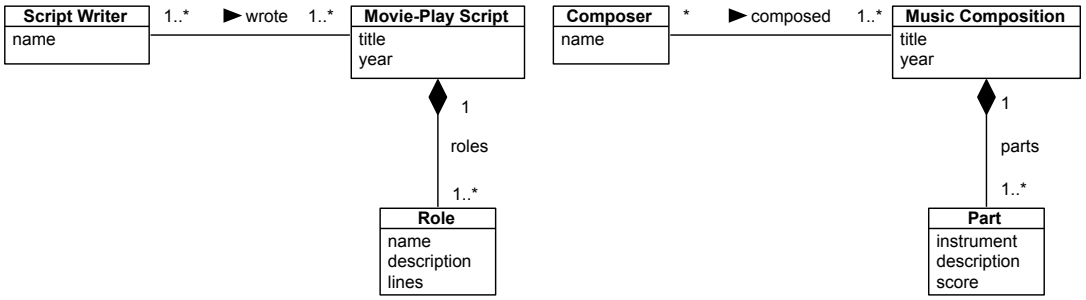


**Example** What is a college "course"? A close inspection reveals every course has two aspects: a general specification that exists in a curriculum, with its number, title, credits, hours, and description; another consisting of actual offerings of course sections that are particular instantiations of the general specification. It is the actual course sections that are taught by instructors, attended by students, and assigned rooms. This observation leads to a model of a course consisting of course sections as its parts. Composition is used as each course section must belong to a unique course and deletion of a course entails deletion of all its course sections. A whole Course object encapsulates data that apply to all its component CourseSection objects. This is an example mainly of sense (7).



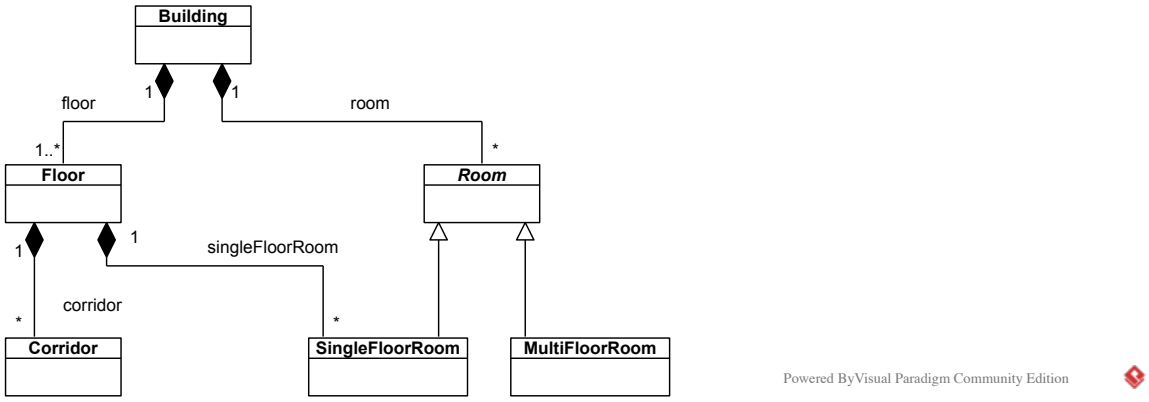


**Example** The following are models of movie/play scripts and music compositions. These are examples mainly of senses (5) and (7). *Movie-Play Script* and *Music Composition* classes might contain data/descriptions of overall guidance/instructions to be observed and interpreted (e.g. by directors and conductors), while *Role* and *Part* might contain similar data/descriptions, primarily intended for actors and players.





**Example** The following is a sample model of buildings. This is an example mainly of sense (1).

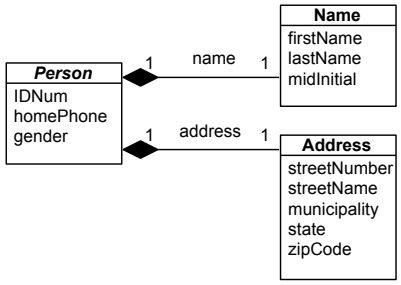




This model illustrates cascade of deletions. When a building object is deleted, all its floors and rooms must also be deleted. In addition, all corridors of the deleted floors must also be deleted.

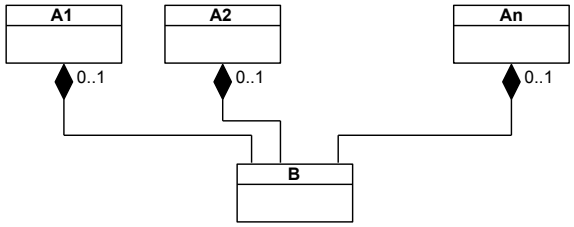
Let $room_S$: Building → SingleFloorRoom be the relation *room* restricted to the range SingleFloorRoom. The logical constraint, $room_S$ = (floor singleFloorRoom), would normally be imposed in the above model.

Another type of frequently used strong composition is a *subordinate attribute class*, which simply groups some of the related attributes of the *owner class* as a strong composition part. This is useful for organizing attributes in a hierarchical structure. The following model has the subordinate attribute classes *Name* and *Address*. The attributes in these classes are said to be *sub-attributes* of *Person*. This is an example mainly of sense (1).



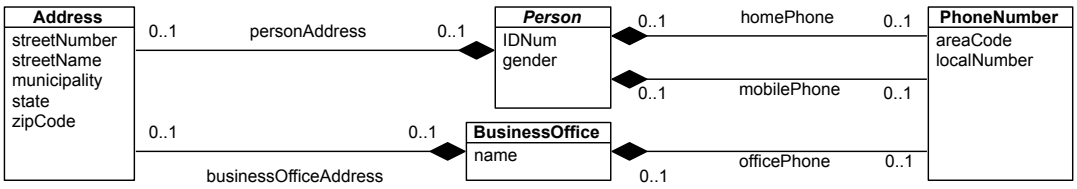



A B-object could be a strong composition part of one of multiple potential whole classes $A_1$, ..., $A_n$.






In this case, the uniqueness of the whole must hold over the union of all the $A_i$: Each B-object may be a part of at most one object in $A_1 \cup \ldots \cup A_n$. In the following example of the subordinate attribute classes PhoneNumber and Address, each PhoneNumber object belongs to at most one person or business office, likewise for each Address object:






A relation R(A, B) should be modeled as strong composition only if it makes sense to regard B as part of A, not just because the formal requirements are satisfied. For example, consider this model of directed networks:



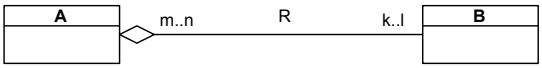

Here the association relations *source* and *target* specify the source and target node of each link. Since every directed link has the unique source and target, the relations *source* and *target* have the multiplicity of 1 on the side of *Node*. The source and target nodes of any link are invariant and do not change. And if any node is deleted, all the incident links would also be deleted in many applications. Yet it would be unsound to regard links as parts of nodes, and so strong composition would not be used in this case for the semantic reason; if necessary, the requirement of deletion of incident links may be specified in the text documentation area.

The relation composition $R_1 \cdots R_n$ is a strong composition relation if each $R_i$, $1 \le i \le n$, is a strong composition relation, demonstrated as follows. First consider $R_1R_2$. Since $R_1$ and $R_2$ are strong compositions, for each object $c$ there is at most one $b$ such that $R_2(b, c)$ and at most one $a$ such that $R_1(a, b)$. Hence there is at most one $a$ such that $R_1R_2(a, c)$ for each $c$. To show $R_1R_2$ cannot change the whole object of any $c$, suppose $R_1R_2(a, c)$ holds at time $t_1$ and $R_1R_2(a', c)$ holds at time $t_2$. Then at $t_1$, there is $b$ such that $R_1(a, b)$ and $R_2(b, c)$, and at $t_2$, there is $b'$ such that $R_1(a', b')$ and $R_2(b', c)$. Since $R_2$ is strong composition, $b \neq b'$ is not permitted, so $b = b'$ must hold. Since $R_2$ too is strong composition, then, $a = a'$ must hold. To show cascade of deletions, suppose $R_1R_2(a, c)$ holds. Then for some $b$, $R_1(a, b)$ and $R_2(b, c)$. Since $R_1$ is strong composition, $b$ must be deleted whenever $a$ is deleted. Since $R_2$ is strong composition, then, $c$ must also be deleted. Thus $R_1R_2$ is a strong composition relation. By induction, therefore, the relation composition $R_1 \cdots R_n$ is a strong composition relation if each $R_i$, $1 \le i \le n$, is a strong composition relation.

**Aggregation Relations (Weak Composition Relations)** An aggregation whole-part relation R(A, B) imposes no specific constraints on the relation. In particular:

- Each B-object may be a part of multiple A-objects simultaneously.
- Each B-object may change its whole object.
- Deletion of an A-object does not necessitate deletion of its part B-objects.

The underlying intuition is that a whole is an aggregation or assemblage of parts with functional roles that can be fulfilled, performed, or served by changeable objects that exist on their own and do not "live and die" with the whole. An aggregation relation is graphically represented by a line between A and B labeled with the relation name R, with a small white diamond at A's end. The ranges of multiplicities on both sides should be shown using the same x..y notation used for association relations.



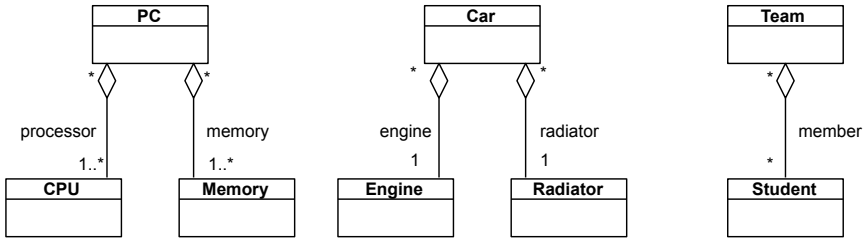



Composition relations are special cases of aggregation relations with the two constraints of the uniqueness of the whole and the dependence of parts' lifetime on the whole's. Therefore it would not be formally incorrect to use an aggregation relation where a composition relation is applicable. Doing so, of course, fails to give important structural and lifetime information captured by the two constraints of composition relations. So aggregation relations should only be used when composition relations are not applicable.

We showed that the relation composition $R_1 \cdots R_n$ is a strong composition relation if each $R_i$, $1 \le i \le n$, is a strong composition relation. If at least one $R_i$ is an aggregation relation while the others are strong composition, $R_1 \cdots R_n$ must be regarded as an aggregation relation because $R_i$ breaks the propagation of the requirements of strong composition.

**Example** Consider the relation processor(PC, CPU) where processor(*p*, *c*) means a PC model *p* uses a CPU model *c*. A CPU model may exist without being a part of any PC model, or it may be used in one or more PC models. Likewise for the relation memory(PC, Memory). Also, deletion of a PC model does not necessitate deletion of the CPU model or the memory model it uses. Analogous aggregation relations can be used between the class of car models and the classes of engine and radiator models. The use of aggregation relations is suitable for modeling components of mass-produced machines and apparatuses.

**Example** Consider the relation member(Team, Student) where member(*t*, *s*) means varsity team *t* has student *s* as member. A student may be members of any number of teams, and deletion of a team does not necessitate deletion of its student members. Aggregation relations are also suitable for modeling groups, teams, or committees of changeable members.

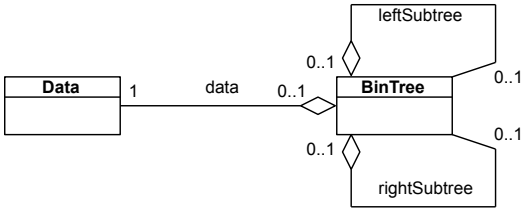





The difference between association and whole-part relations is a matter of the relations' semantic meaning and not a formal one. In some cases, a binary relation may be legitimately regarded as both association and whole-part. For example, consider the relation employ(Office, Employee) where employ(*o*, *e*) means office *o* employs employee *e*. This relation may be regarded as association between two independent classes, or as a whole-part relation where, organizationally, each employee is a part of an office. In the latter case, since an employee may work for more than one office at once and deletion of an office does not entail deletion of its employees, the relation should be modeled as aggregation rather than composition. A case like this is largely a matter of perception, and the choice will be up to the database designer.

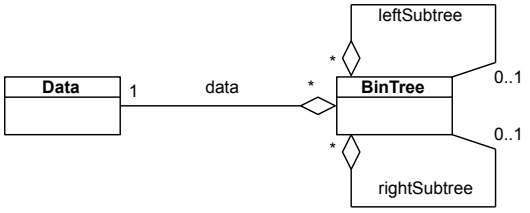
Whole-part structures can be *recursive* in the sense that a whole object of a class A may have A-objects as its parts, sub-parts, etc. Recursive whole-part relations can be aggregation or composition, depending on whether the requirements of composition are met.

**Example** Consider a class of binary trees, with each tree having three components: a data object contained in the root node of the tree, the left subtree, and the right subtree. The following model uses aggregation relations.



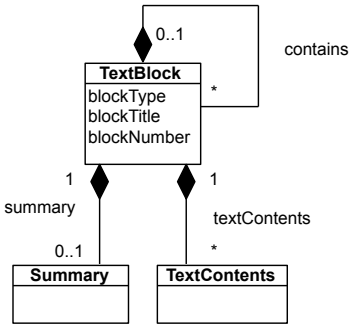



In some applications, a data object and subtrees might be shared by multiple trees:



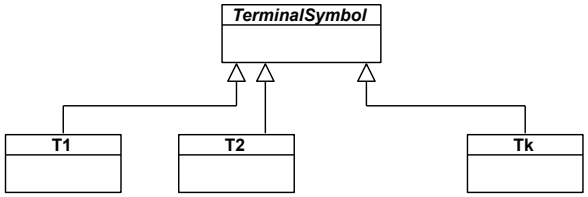



**Example** Document texts normally have hierarchical structures of *text blocks* of various types like parts, chapters, sections, subsections, ..., paragraphs. This hierarchy can be modeled as follows.





**Example** Syntax (parse) trees of context-free grammars can be modeled as follows. Let ⟨V, T, P⟩ be a context-free grammar where V is a set of variables, T is a set of terminal symbols, and P is a set of production rules. We create the abstract class of terminal symbols and its subclasses representing the terminal symbols in T:

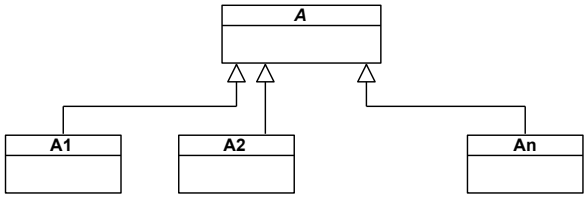




For the production rule of each variable $A \in V$

$$A \to \alpha_1 \mid \cdots \mid \alpha_n,$$

create an abstract class A together with its subclasses $A_1, \ldots, A_n$ representing $\alpha_1, \ldots, \alpha_n$. The subclass $A_i$, $1 \le i \le n$, has the strong-composition and weak-composition parts representing, respectively, the variables and terminal symbols occurring in $\alpha_i$:

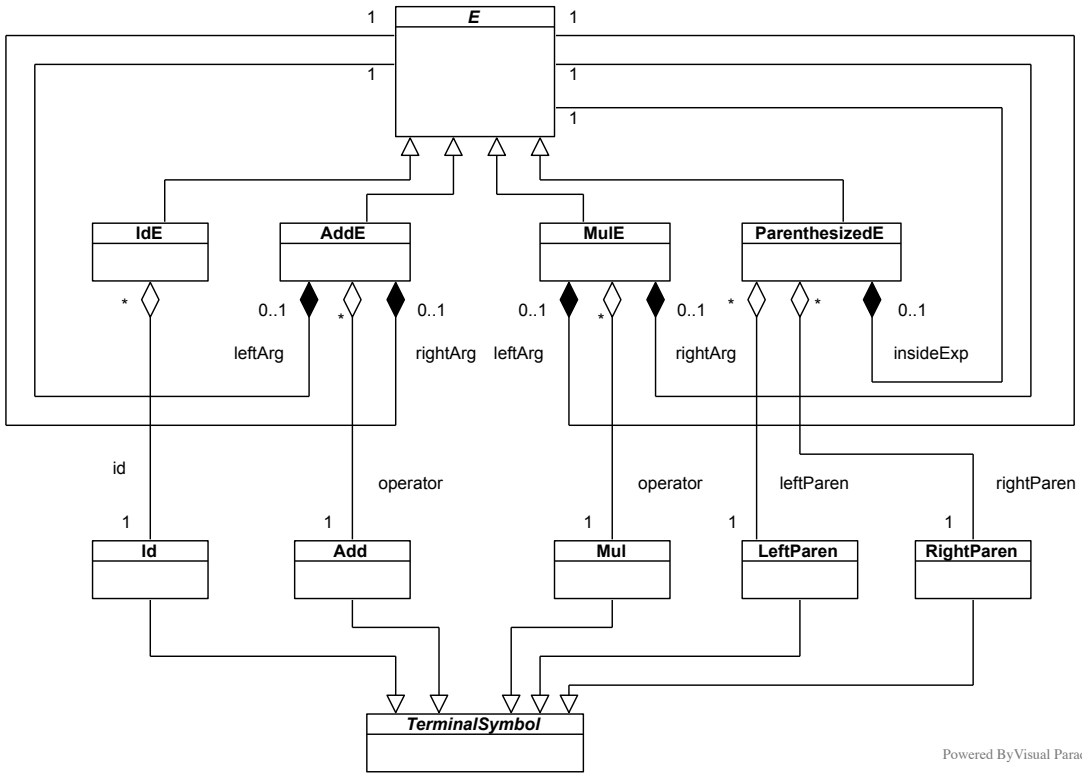




Consider, for example, the following context-free grammar for arithmetic expressions where V = { E }, T = { id, +, *, (, ) }, and the production rule:

$$E \to \text{id} \mid E + E \mid E * E \mid ( E )$$

The syntax trees of this grammar can be modeled as follows:

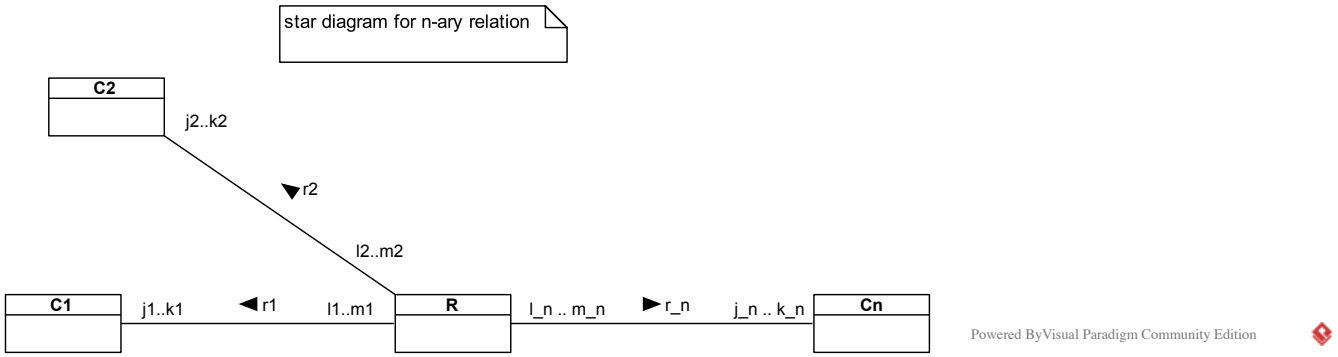


Every E-object is a strong-composition part of at most one object in AddE ∪ MulE ∪ ParenthesizedE.

Sometimes the choice between aggregation and composition is made in consideration of functionality requirement of the database, that is, what information to keep in the database. In the binary tree example above, the relevant requirement might be that when a certain subtree is deleted, its Data object in the root and/or the left and right subtrees should also be deleted. In this case, strong composition should be used. In the text block example, on the other hand, we may want to keep low-level blocks (e.g., sections) even when high-level blocks (e.g., chapters) are deleted, so that we can later reassemble the low-level blocks to form new high-level ones. In this case, aggregation should be used.

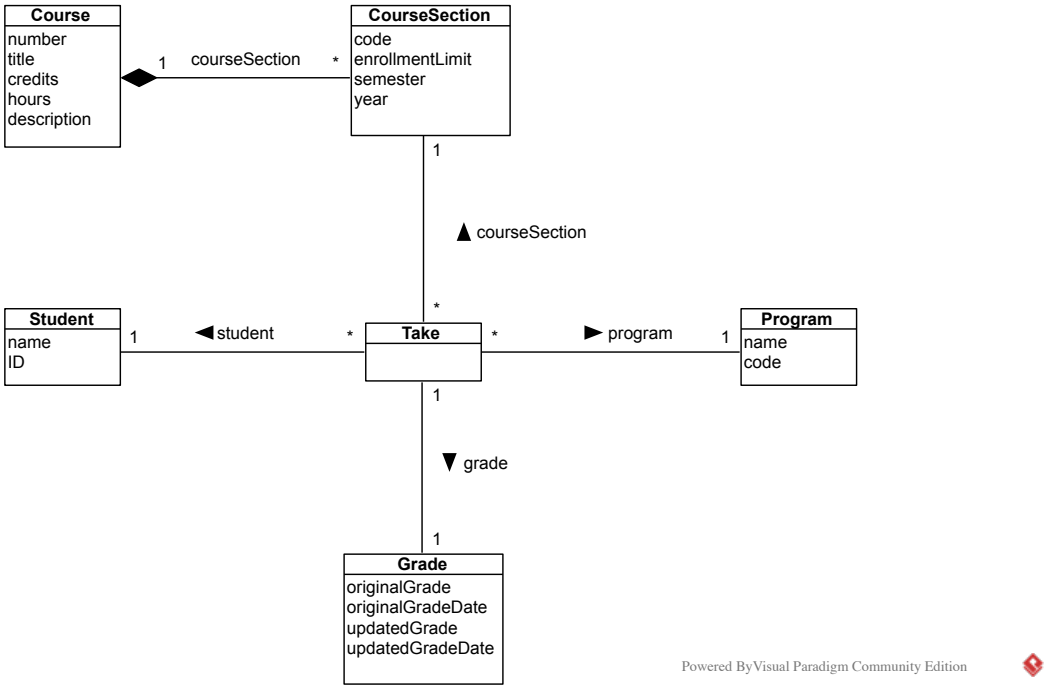
## 2 N-ary Relations

For OODBs, an *n*-ary relation $R(C_1, ..., C_n)$ over classes $C_1, ..., C_n$, for any $n \geq 2$, can be modeled by a *class*, labeled R, together with binary relations $r_i(R, C_i)$, $1 \leq i \leq n$.

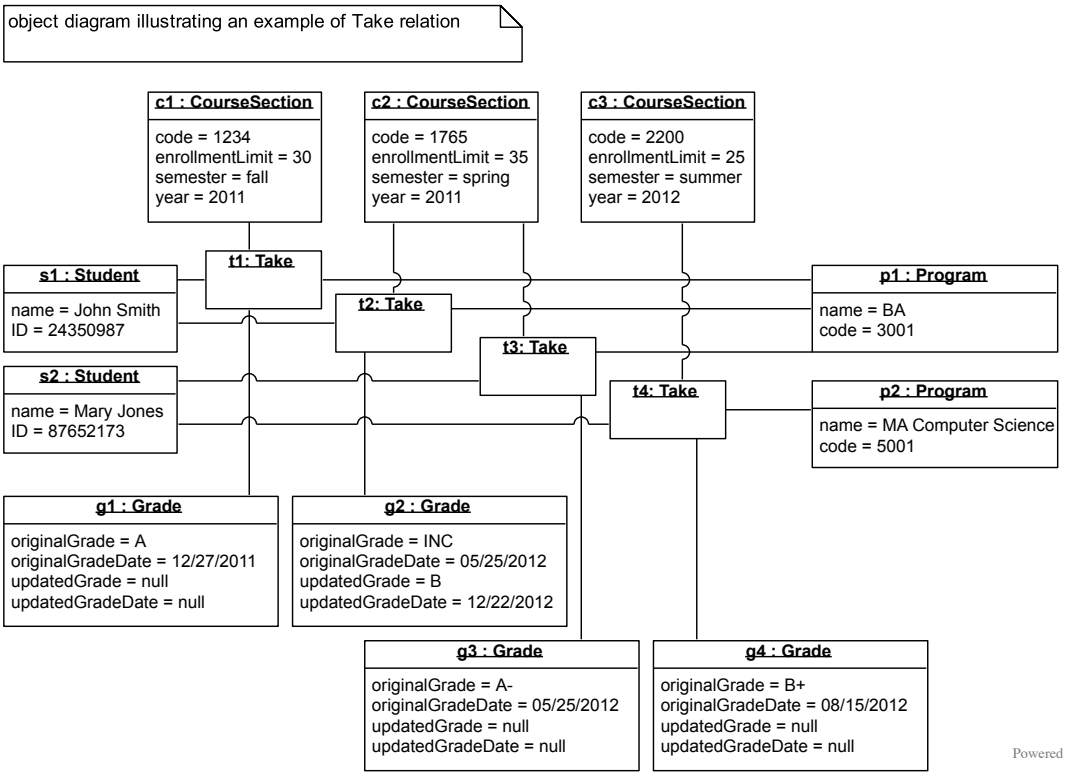



The class R's objects may represent the *n*-tuples comprising the relation R with the semantics of $r_i(r, c)$ meaning that $C_i$-object *c* is the i-th component of the tuple *r*. In this case the multiplicity of each $r_i$ on $C_i$'s end is 1, because every tuple has a single object at each position. In some relations i-th component may be absent temporarily or permanently in some tuples; in this case the multiplicity of 0..1 should be used for $C_i$'s end.

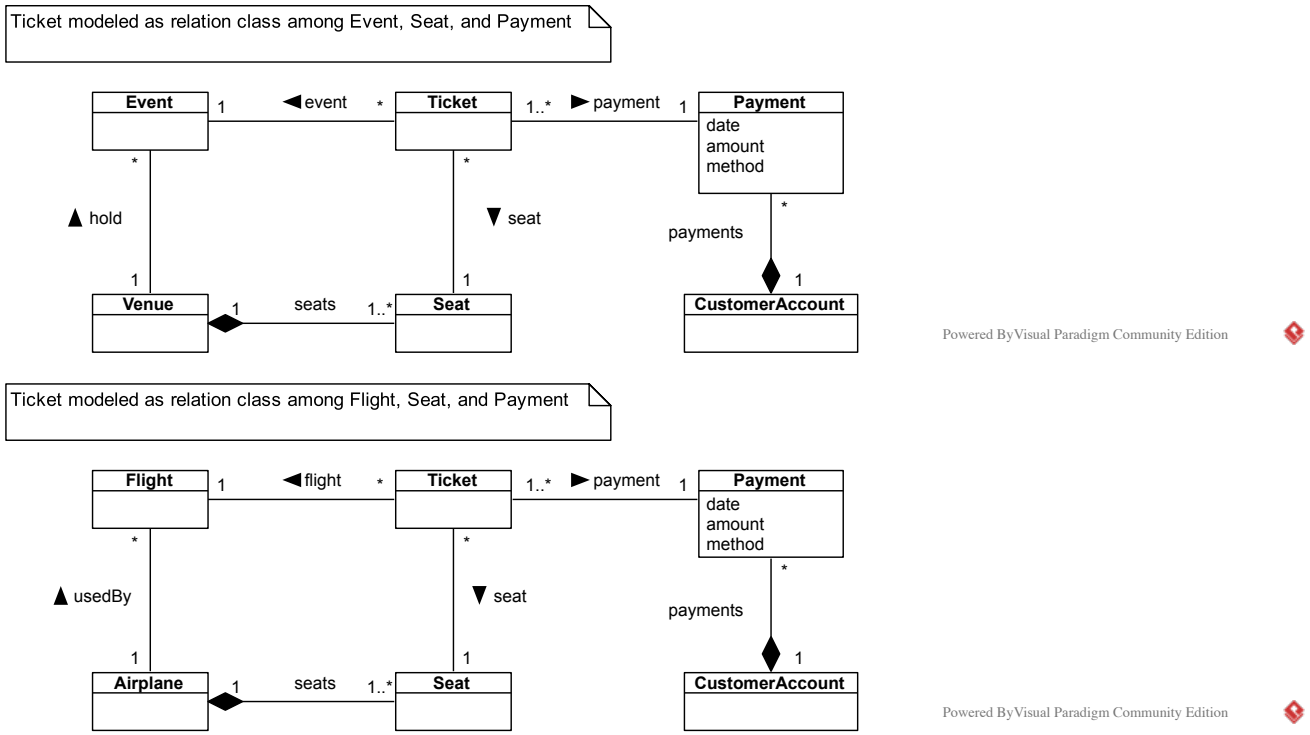
**Example** The following diagram models the 4-ary relation Take(Student, CourseSection, Program, Grade) where Student, CourseSection, Program, and Grade are, respectively, the classes of students, course sections, programs, and grades in a certain college. The semantics of Take(*s*, *c*, *p*, *g*) is that student *s* took course section *c* in program *p* receiving grade *g* (here we use all letter grades, including incomplete, absent, withdraw, etc.).



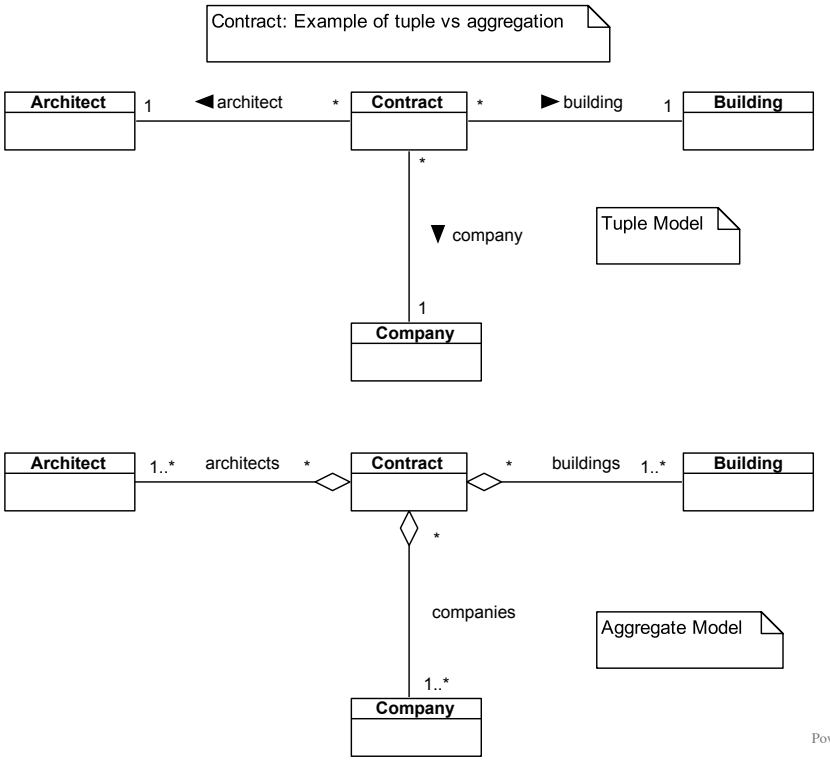

Each Take object represents a 4-tuple ⟨*s*, *c*, *p*, *g*⟩. Every Grade object belongs to a unique Take tuple, hence to a unique Take object, although some Grade objects may have the same attribute values by coincidence. If the same Grade object were shared by two or more Take objects, an update of the Grade object would affect all the sharing Take objects involving different students and course sections. Clearly this must be avoided, hence the 1:1 correspondence between Grade and Take. The Grade class may be regarded as a subordinate attribute class of Take, so it makes sense to use strong composition for grade(Take, Grade). If a Take object is deleted for any reason, its Grade object should also be deleted, but the associated Student, CourseSection, Program objects should not be deleted.



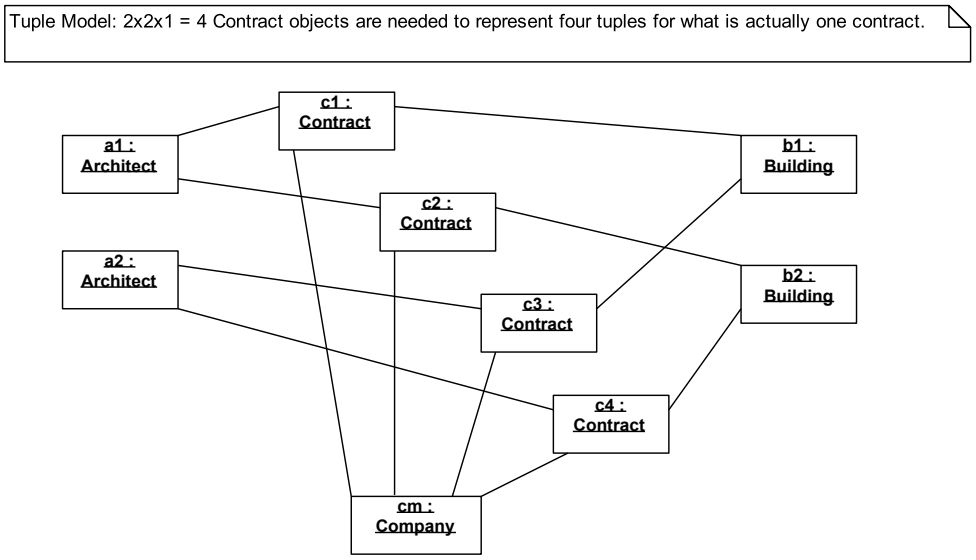

Another example of relation class is that of electronic tickets.

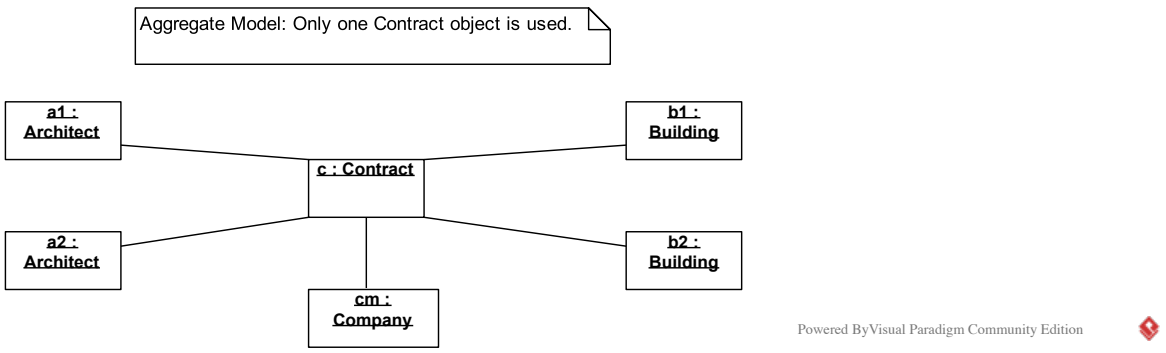



An object in the class R need not represent a tuple of the relation R, and may relate by $r_i$ to more than one object in $C_i$. Consider, for example, the 3-ary relation contract(Architect, Company, Building) among the classes of architects, companies, and buildings. A triple ⟨*a*, *c*, *b*⟩ is in the relation if and only if architect *a* has signed a contract with company *c* to design building *b*. In the tuple model below, the objects of Contract class represent the triples in the relation. However, a single contract may involve multiple architects, multiple companies, and/or multiple buildings. This possibility is better accommodated by the aggregate model which views a contract not just as a relation, but also as an aggregate with architects, companies, and buildings as components.




A contract is a relation, but it is also an object with its own identity. The aggregate model captures a contract more naturally without dividing the collective information into tuples. In contrast, the tuple model uses a set of disconnected tuples to represent a single contract among multiple architects, companies, and buildings. This fails to capture their binding by the same contract, and an attribute of the Contract class, like contract ID, would be necessary to record the common binding and the contract's identity. The following object diagrams illustrate the difference by an example contract involving architects *a1*, *a2*, buildings *b1*, *b2*, and company *cm*.



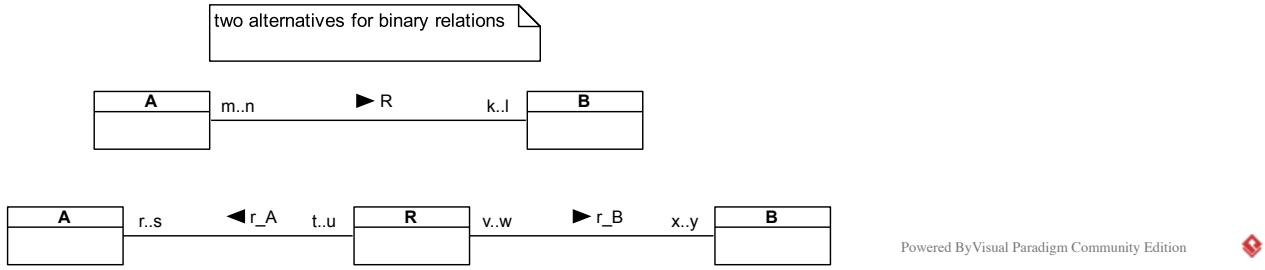



In general, suppose we have an *n*-ary relation $R(C_1, ..., C_n)$ over classes $C_1, ..., C_n$. As in the above contract relation example, a single aggregate object may properly model each *n*-ary relation instance among $k_1$ objects in $C_1$, …, $k_n$ objects in $C_n$. The corresponding tuple model would need $k_1 \times \cdots \times k_n$ disconnected tuple objects. Thus the aggregate model can offer benefits not provided by the tuple model.

Note that any *n*-ary relation $R(C_1, ..., C_n)$ is inherited by the descendant classes of each $C_i$. That is, a relation instance $R(x_1, ..., x_n)$ may hold as long as $x_i$ is a direct object of $C_i$ or is an object of one of its descendant classes, for any i: $1 \le i \le n$.

## 3 Two Alternative Modeling Methods for Binary Relations

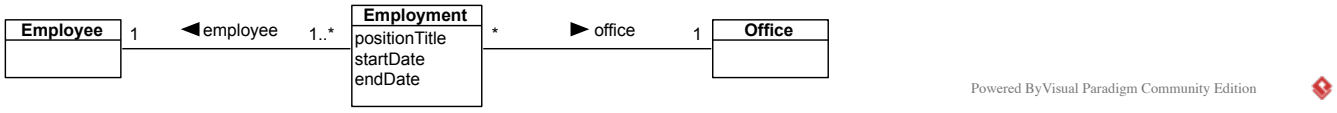
Binary relations can be modeled without using relation classes (as described in Section 1) or by relation classes (as described in Section 2).



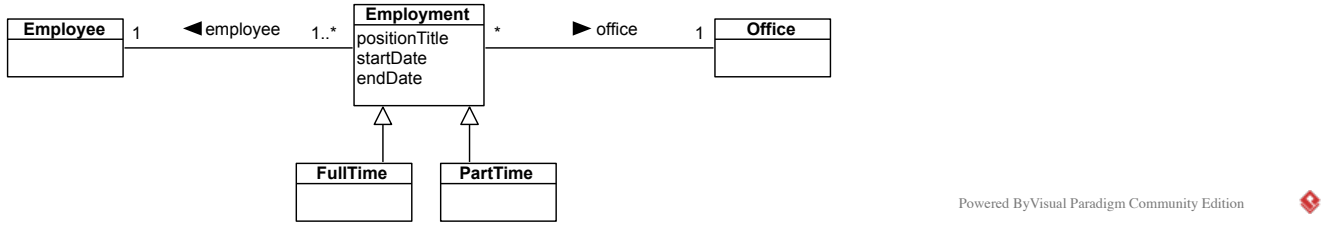

The first method is comparatively straightforward and leads to a simpler implementation. The relation R is implemented by a set-type field in the class A holding the set $R{\downarrow}B(a)$ for each $a \in A$. Optionally, the class B may have a field holding the set $R{\downarrow}A(b)$ for each $b \in B$ implementing the inverse $R^{-}$. In case $|R{\downarrow}B(a)| \le 1$ always holds for each $a \in A$, R is implemented in A by a field of class type B without using sets; likewise for the optional field in B. Generally, this method leads to simpler forms of queries involving the relation R.

The second method is conceptually more complex and leads to a more complex implementation since we need to deal with the relation class R along with the two relations $r_A$ and $r_B$. However, modeling a binary relation by a class enjoys a few advantages.

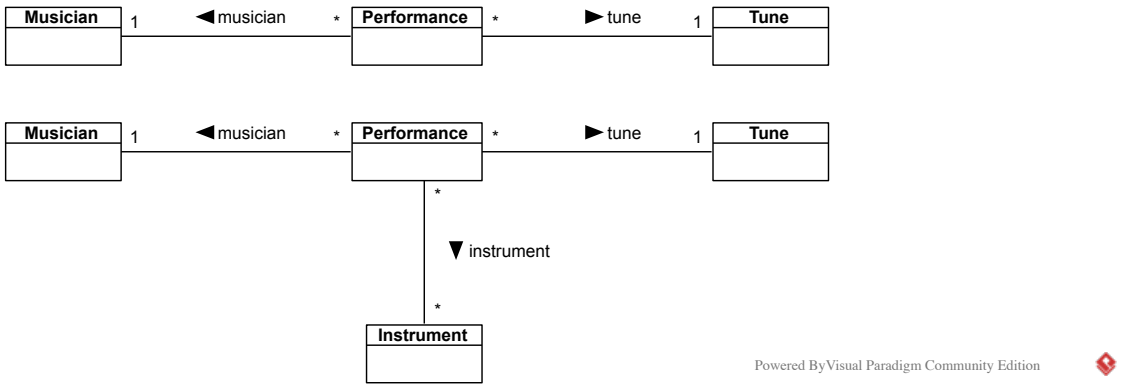
- The relation class R can have its own attributes and functions. The diagram below models the binary relation *employment* between Employee and Office, with the relation class *Employment* incorporating the attributes: positionTitle, startDate, and endDate. The *Employment* class may have functions to return the values of these attributes and the duration of the period from startDate to endDate.




- The relation class R itself can be specialized to descendant classes, forming a specialization hierarchy of its own. For example, the *Employment* class may be specialized to subclasses depending on employment types like full time and part time employment. This would be useful if different employment types require different kinds of information about salary structure, fringe benefits, etc.




- The use of a relation class facilitates incremental additions of its arguments. For example, consider the relation performance(Musician, Tune) where performance(*m*, *t*) means musician *m* performed on recorded tune *t*, which is modeled as a relation class. Later an extra argument, *Instrument* class, may be added to form the 3-ary relation performance(Musician, Instrument, Tune) where performance(*m*, *i*, *t*) means musician *m* performed musical instrument *i* on tune *t*.



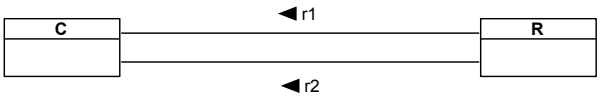

- The relation class R itself can participate in other relations. For example, we may create the relation canAccess(Person, Employment) where canAccess(*p*, *e*) means person *p* has access permission to read Employment-object *e*.

These advantages of relation classes are not limited to binary relations and apply to *n*-ary relations in general.

A binary relation R(C, C) is *symmetric* if

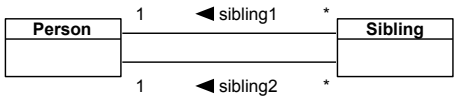
$\forall x,y \in C\ (\ R(x, y) \Rightarrow R(y, x)\ )$.

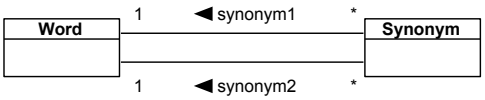
A symmetric relation can be modeled by a relation class R:





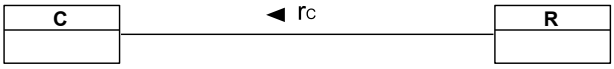

For example:



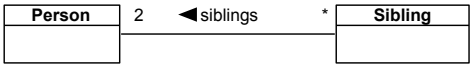


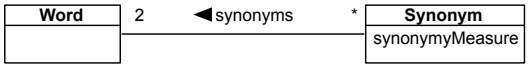

But the symmetry of R makes the use of two relations $r_1$ and $r_2$ superfluous, so a better model is:





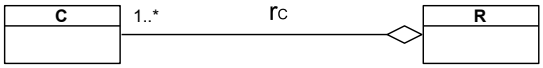

For example:



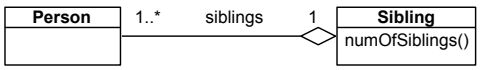


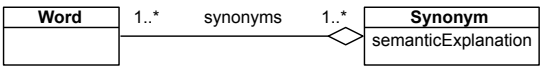

These are examples of tuple models. If there are x siblings, x(x−1)/2 unordered pairs need to be related by the *siblings* relation to as many Sibling objects, and there is no immediate way to find all x siblings. In case more than two objects mutually relate to one another by R, the following aggregate model would be more suitable:






with the requirement that for each $r \in R$ and for any distinct x, $y \in r_C{\downarrow}C(r)$, R(x, y) holds. For example:

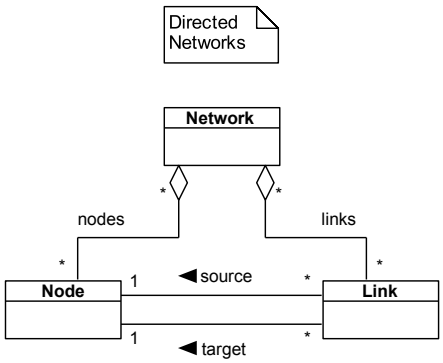


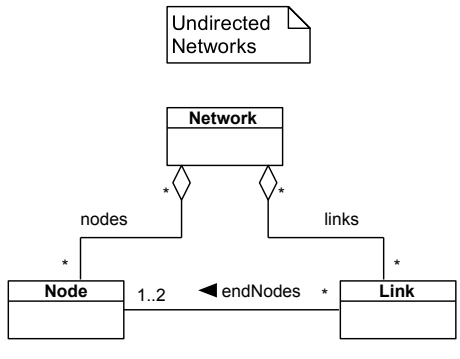






Here, all x siblings are parts of one Sibling aggregate object. The multiplicity of the *Synonym* end of the *synonyms* relation is 1..* as a word may belong to more than one synonym group determined by its different meanings.

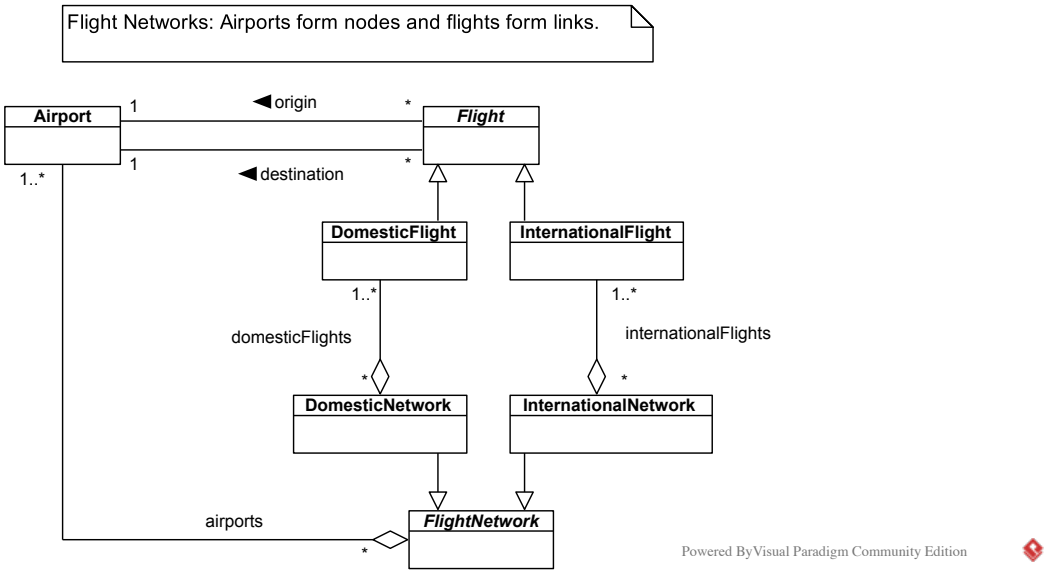
*Networks* of nodes and connecting links are broadly applicable data models, e.g., computer networks, telephone networks, and transportation networks such as airline networks, train networks, roads and highways. The links of a network can be regarded as a binary relation between nodes and modeled to advantage by a relation class. The following are example models of directed and undirected networks.






In these models the Link class can have attributes to hold data associated with links, such as data transmission rates of links in computer and telephone networks, and distances between two nodes in airline and road networks. Link objects can also participate in other relations. If links represent segments of roads and highways, for example, we may create two relations from the Link class to a class of states and to a class of cities, respectively indicating which road/highway segment traverses which state and city. The Link class may be specialized to subclasses. In airline networks where links represent flights, the Link class may have subclasses of domestic and international flights.




## 4 Standard Identifiers Convention

The UML community uses the following standard convention for identifiers:

- Class names should begin with uppercase letters; if a class name is a noun, it should be in singular form.
- Relation names, class attributes, and function names should begin with lowercase letters.

We also emphasize the importance of providing text documentation whenever necessary. The names of classes, attributes, and relations should be made as self-explanatory as possible, but names can describe only so much of their meaning. When names are insufficient, the intended meaning of classes, attributes, and relations should be described in clear English. UML tools provide text documentation areas for this purpose. Remember, documentation and communication are important functions of models and specifications.