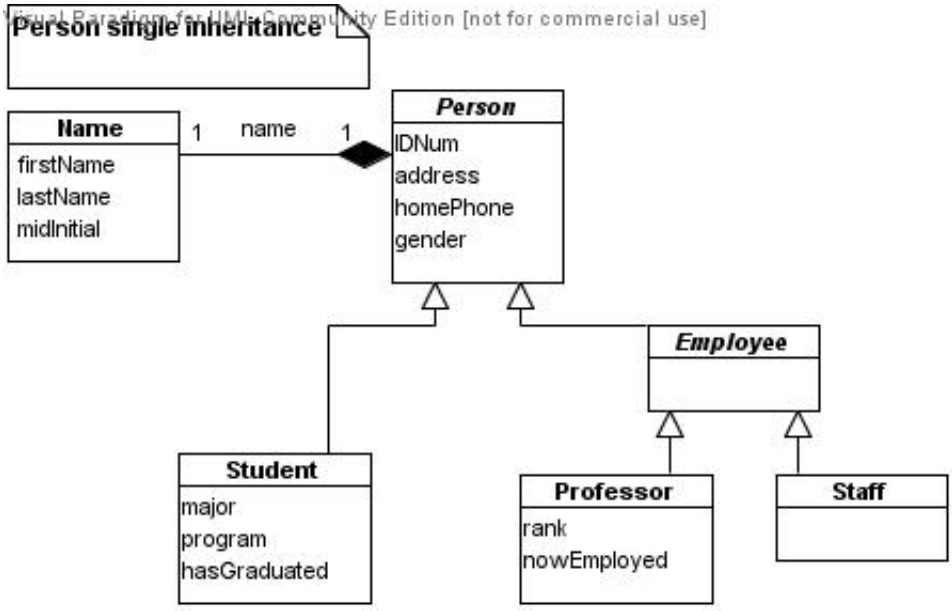
CS 332/780: Object-Oriented Databases Spring 2018
Course Notes #6
Keitaro Yukawa
Department of Computer Science
Queens College, CUNY

Multiple Inheritance

Multiple inheritance lets us derive a subclass D from multiple superclasses C_1, ..., C_n. The subclass D inherits all the attributes, functions, and relations of the *n* superclasses, and captures a class of objects that have all the characteristics of the superclasses. Every D-object is also a C_i-object for all i, $1 \le i \le n$, and so the extent of D is a subset of the intersection of the extents of all C_i's.

The following is a UML model of persons associated with a college (taken from the college model on the last page of Course Notes #4).



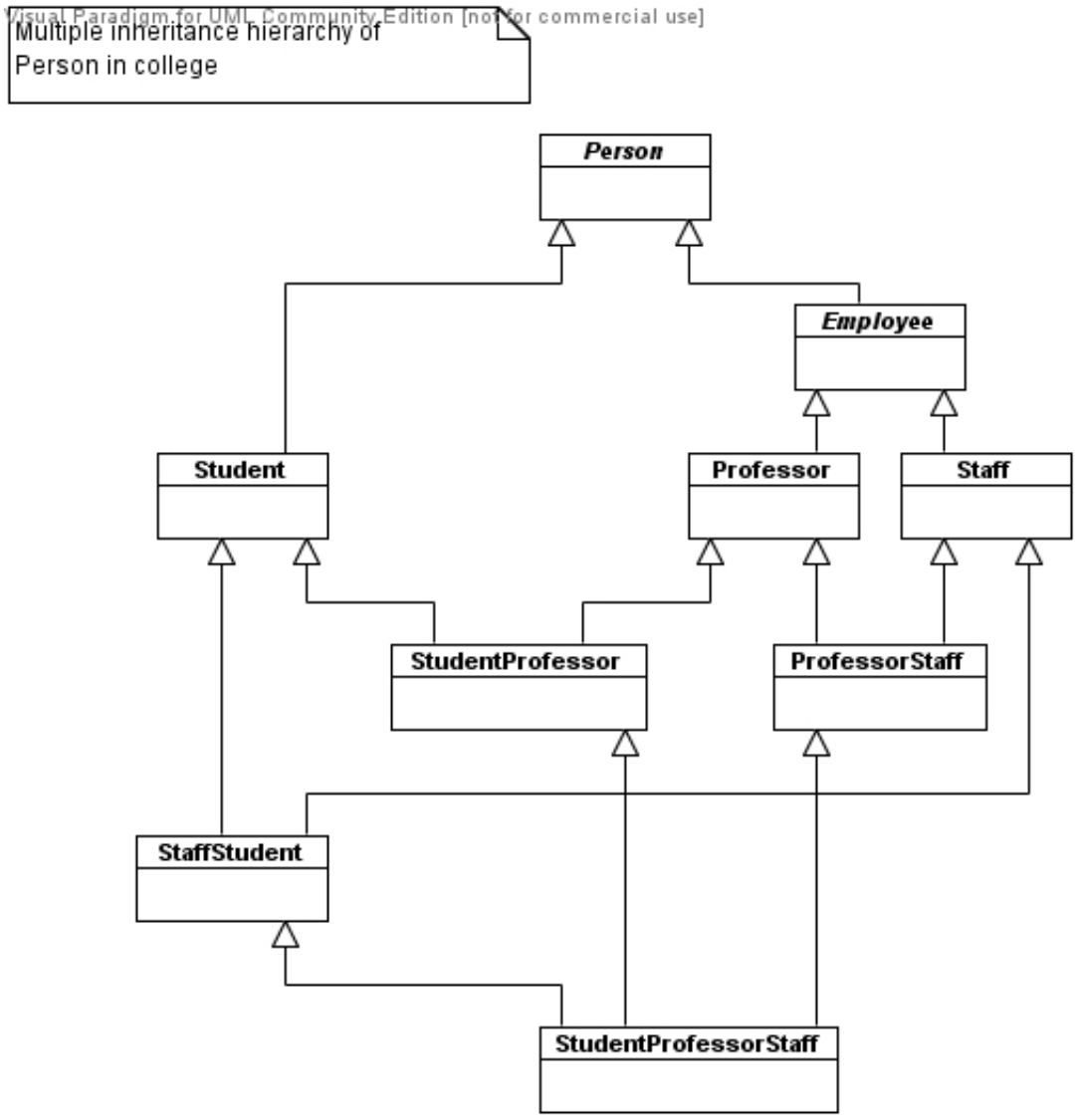
A person in a college may belong to any two or all three of Student, Professor, and Staff. For example, a professor may have been a student and/or a staff member, and a student may have been a staff member, during overlapping or non-overlapping periods. In the above UML model, a student who is also a staff member would be represented by two objects, one in Student and the other in Staff. A person belonging to all three classes would be represented by three objects. Situations of this kind cause the "data duplication" problem and suffer from the following undesirable effects.

- Each of the multiple objects representing the same person has its own copy of the common attributes (name, IDNum, address, homePhone, gender). This is a waste of storage space, and there is the burden of ensuring the consistency: the objects must have the same values for the five attributes at all times.
- No explicit relation exists between these objects to indicate they represent the same person; this information needs to be recovered from IDNum.
- IDNum is not a key attribute for the Person class, since it fails to identify Person objects uniquely. (IDNum can be made a key for Person by assigning distinct IDNum values to all Person objects, but then it is no longer possible to determine by IDNum if a Student object and a Staff object, for example, represent the same person in reality.)
- Fundamentally, using multiple disconnected objects to represent a single object with unique identity

in reality is a violation of a principle of the object-orientation concept.

These problems could be alleviated by promoting the IDNum, address, homePhone, gender attributes to objects and making the multiple objects for the same person share them. Also, name: Person → Name relation would have 1..*, 1 multiplicities due to sharing and no longer can be strong composition. Application programs must ensure that when a person acquires a new status, for example when a student becomes a staff member, a newly created object shares the five attribute objects.

Multiple inheritance eliminates these problems by deriving four subclasses: StudentProfessor, ProfessorStaff, StaffStudent, and StudentProfessorStaff, each inheriting from the classes indicated by its name.



The class StudentProfessorStaff should inherit from {StudentProfessor, ProfessorStaff, StaffStudent} so that it inherits the attributes, functions, and relations that may be defined in these three classes, rather than from {Student, Professor, Staff}, {StudentProfessor, Staff}, etc. For example, suppose a certain numerical adjustment needs to be made to the salaries of professors who are also staff members. this can be naturally stored as an attribute or function of the ProfessorStaff class. This should be inherited to the StudentProfessorStaff class. Likewise, tuition adjustments for students who are also professors or staff members are stored as features of StudentProfessor or StaffStudent. These are examples of *emergent properties*: A subclass D derived from C_1, ..., C_k may have new properties which are not present in any of C_1, ..., C_k but which emerge from the fact that D is all of C_1, ..., C_k.

Each of the four subclasses has one copy of the attributes (name, IDNum, address, homePhone, gender) inherited from Person (because of the name-clash resolution rule we adopted in §4.2 of Course Notes #3). When a person acquires an additional status, the person moves from the current class to one of its descendants. This means the person's object becomes a direct object of a descendant class and an indirect object of the current class. For example, when a staff member becomes a student, that person's object becomes a direct object of StaffStudent and an indirect object of Staff. (In OODBs, this would be implemented by deleting the person's object from the current class and creating a new object in the descendant class. OODBs might provide a built-in operation to move an object to a descendant class preserving its object identity/reference value.)

Multiple inheritance, however, potentially causes the problem of combinatorial explosion in the number of subclasses. Assume we have *n* classes. The total number of combinations of *k* classes, $2 \le k \le n$, that can spawn subclasses by multiple inheritance is:

$C(n, 2) + C(n, 3) + ... + C(n, n) = 2^n - n - 1 = \Theta(2^n)$,

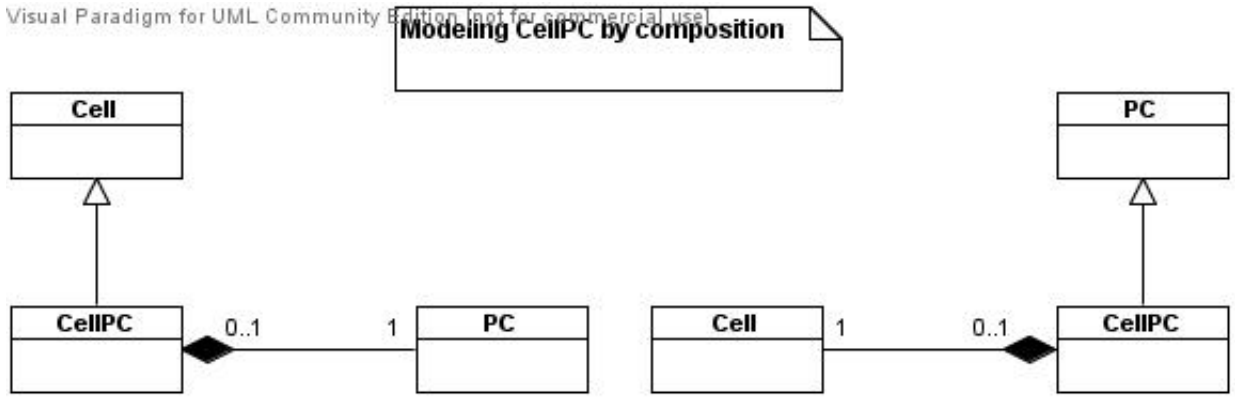
where $C(n, k)$ is the binomial coefficient, i.e., the number of ways to choose *k* things from *n* things. For $n = 5, 10, 20$, the numbers are 26, 1013, and 1048555, respectively. In reality, a smaller number of subclasses would be needed, but it could still become very large. A large multiple inheritance hierarchy can be difficult to manage and comprehend. When there is a danger of combinatorial explosion, we may consider the alternative method of using composite structure with whole-part relations – this method eliminates combinatorial explosion and will be described in the next section.

Another crucial factor influencing the extent of use of multiple inheritance in models is its availability in the database system that will implement the models. If the implementing database system does not support multiple inheritance, multiple inheritance hierarchies in the model have to be somehow mapped to structures supported by the system (e.g., using the composite-structure method described in the next section). In this case, naturally, multiple inheritance would be used with caution. Java Data Object is an example of this, for it is based on Java and does not support multiple inheritance.

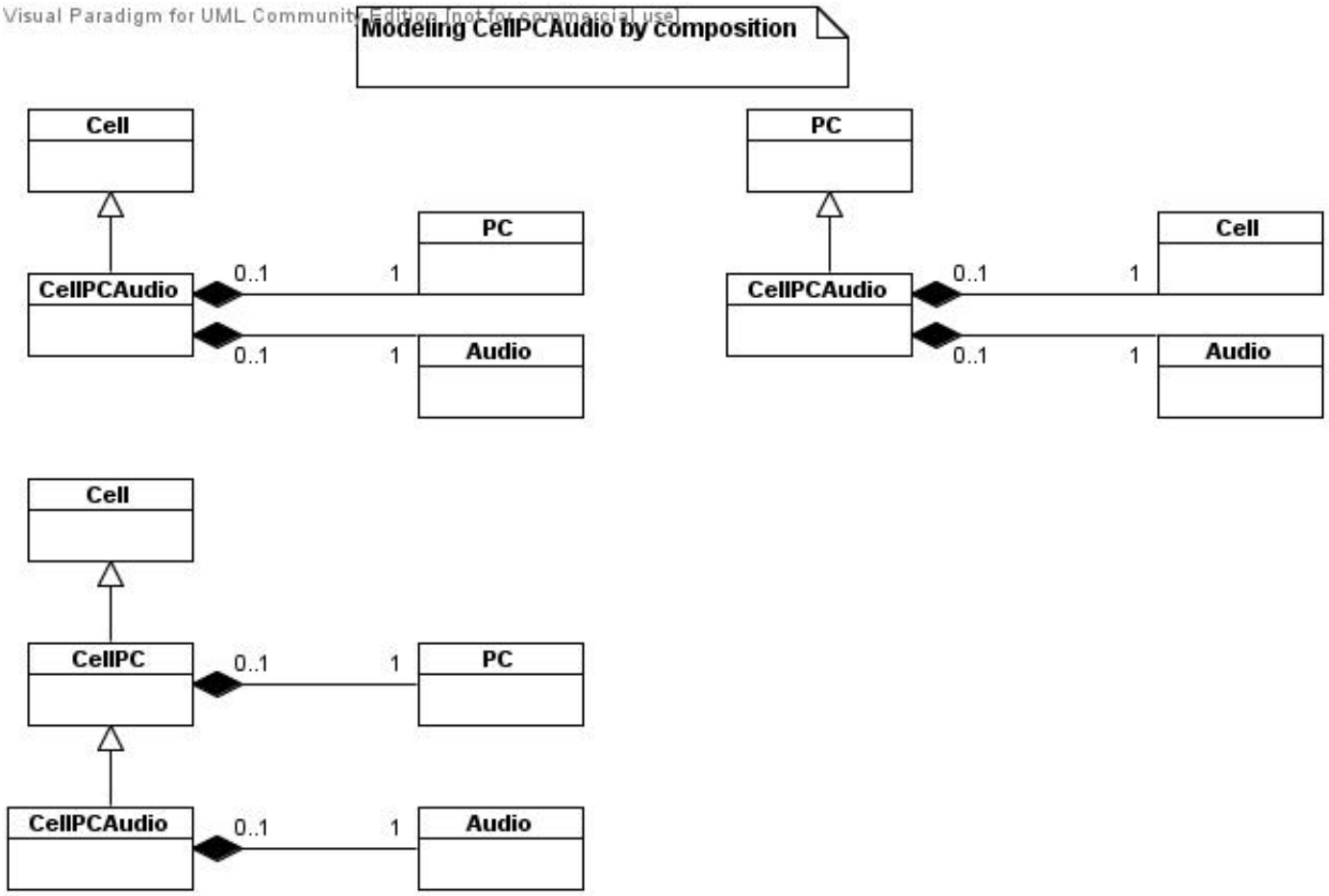
Composite Structure as Alternative to Inheritance

We describe a method of using composite structure with whole-part relations as an alternative to inheritance. Although automatic inheritance of class features and inheritance polymorphism are absent, this method can simulate subclasses in multiple inheritance hierarchies while eliminating potential combinatorial explosions in the number of subclasses.

We begin with a relatively ad hoc method. Suppose we have three classes of cell phones, pocket PCs, and pocket audio players. By multiple inheritance we could derive up to four subclasses of devices combining two or all of the three categories. Alternatively, we can retain one inheritance relation only, and replace the other inheritance relations by strong composition relations. For example, devices combining cell phones and pocket PCs can be modeled by one of the following two:



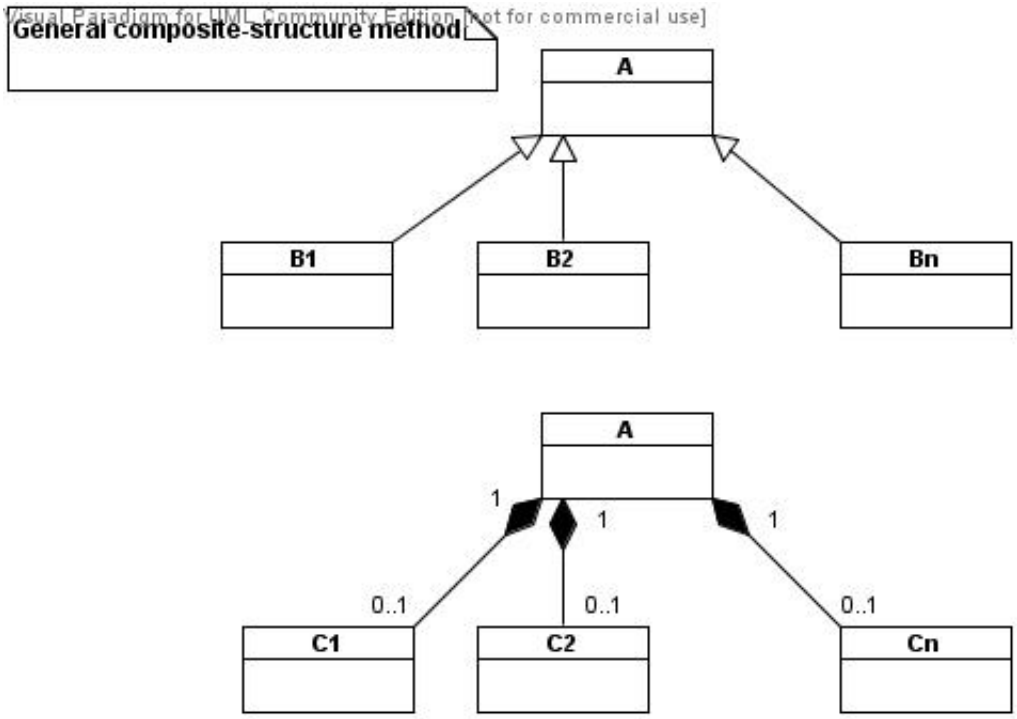
Devices combining all three categories can be modeled by one of the following three (there are other ways):



These models are ad hoc because there are many possible ways of combining inheritance and composition relations and a choice would have to be made on a case-by-case basis or even arbitrarily. Another problem concerns the handling of attributes common to all of the Cell, PC, and Audio classes, like *product name*, *price*, and *weight*. Presumably, such attribute values are only stored in the root class while the same attributes of the other two classes connected by strong composition are left null. This type of ad hoc model may work adequately for two or three subclasses, but will become unmanageable for a larger number of subclasses.

Replacing inheritance relations by composite relations requires the use of so-called *delegation functions*. For example, consider the model of *CellPC* on the left of the first diagram. The features of *PC* class are not inherited to *CellPC*. To simulate the effect of inheritance, we create in *CellPC* functions that will traverse the composition relation to *PC* and then extract and return the relevant features. For example, suppose *PC* has three attributes *cpuClockSpeed*, *memorySize*, and *hardDiskSize*. We create functions *cpuClockSpeed()*, *memorySize()*, and *hardDiskSize()* in *CellPC* whose task is to traverse the composition relation to the relevant *PC* object, extract the corresponding attribute values, and return them. Delegation functions need to be created for every composition relation introduced. These functions execute the act of "delegation" in that the task of extracting a feature of an object, which should be in that object but isn't, is delegated to the related object that actually has it.

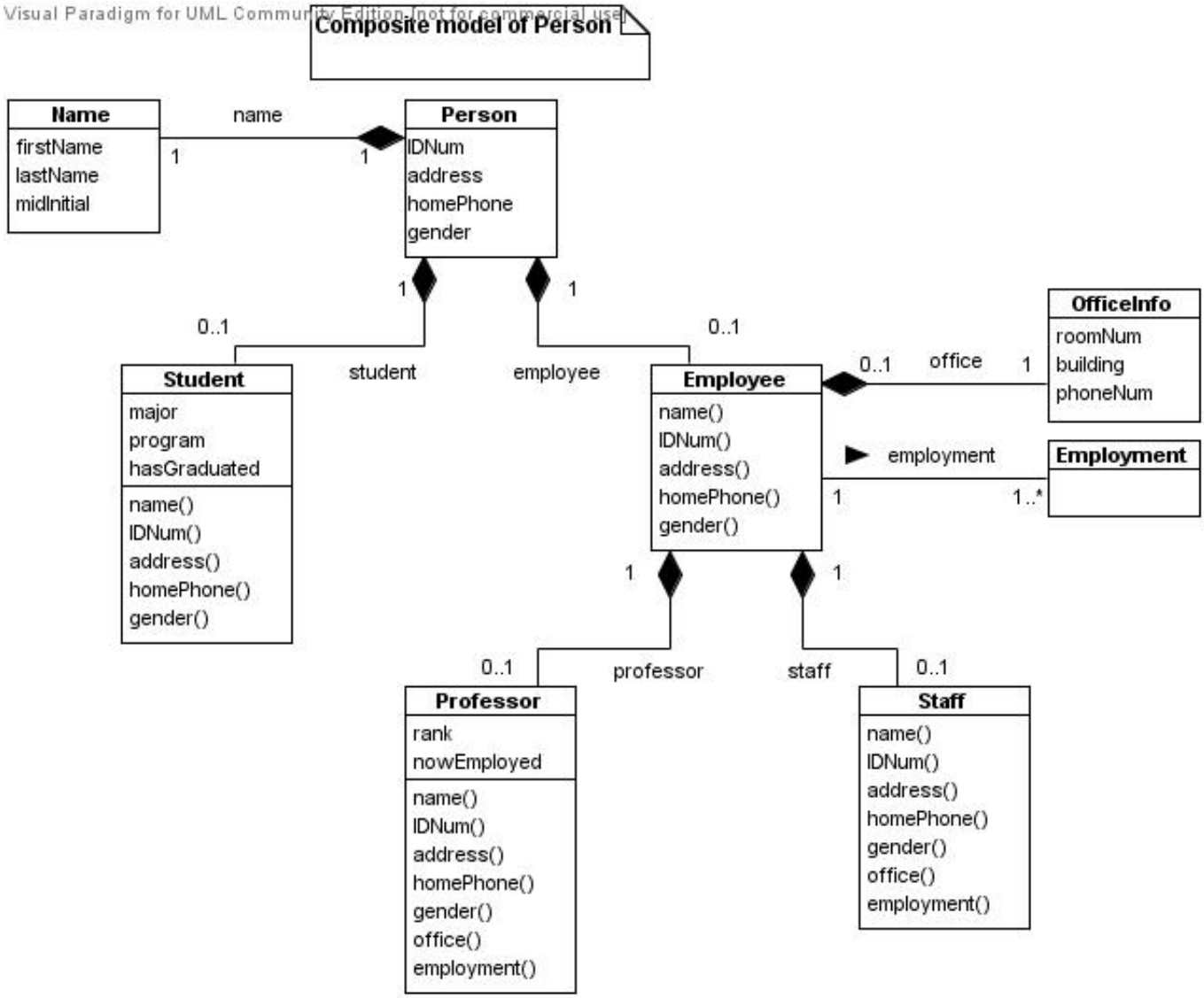
We now describe a more systematic method. Suppose we have a class A and envisage its *n* subclasses B_1, ..., B_n. We create *n* component classes C_1, ..., C_n in place of B_1, ..., B_n, together with strong composition relations $c_i(A, C_i)$, $1 \leq i \leq n$. The multiplicity ranges of each c_i are 1, 0..1.

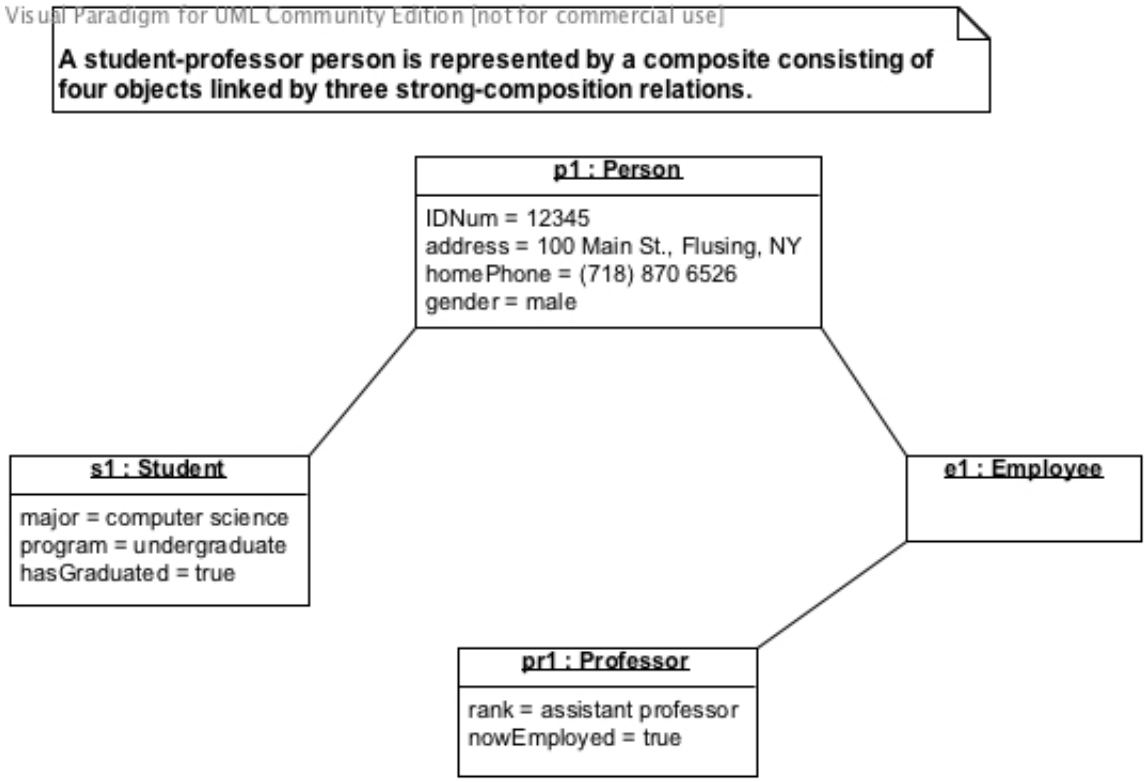


In each C_i we include the attributes, functions, and relations specific to B_i and not inherited from the parent class A. Each B_i-object is then represented by a composite of two objects: an A-object and a C_i-object connected by the composition relation c_i. The use of strong composition is appropriate since every C_i-object only contains data and functions specific to B_i, hence its existence is meaningful only if connected to the corresponding A-object containing the data and functions common to all C_i's. A lower subclass that would be derived by multiple inheritance from a collection *S* of B_i's is represented by the composite of an A-object with the C_i-objects corresponding to the B_i in *S*. All the C_i need the same set of delegation functions to extract A's features that would be inherited to the B_i. A merit of this method is that there is no need to create any lower subclasses that would be derived by multiple inheritance, hence avoiding combinatorial explosion altogether. All of the $2^n - n - 1$ potential subclasses can be simulated by composite objects of A and appropriate choices of C_i's. Turning the multiplicity of C_i off (0) or on (1) lets us represent all 2^n combinations of C_i, including a choice of zero C_i or exactly one C_i.

Here is a useful metaphor: When there is a class A of agents or entities capable of assuming *n* different roles/functions, they can be modeled by the subclasses B_i or, alternatively, the strong composition parts C_i. In the latter case, the classes C_i are called *role classes*.

A composite-structure model for the college Person hierarchy is shown below, including the necessary delegation functions in the role classes Student, Employee, Professor, and Staff. A student who is also a professor would be represented by a composite of four objects: a Person object with two components of a Student object and an Employee object, the latter with one component of a Professor object. A professor who is also a staff member would be represented by a composite of Person, Employee, Professor, Staff objects. A person who is all three would be represented by a composite of Person, Student, Employee, Professor, Staff objects. Note that the Person and Employee classes in this model cannot be abstract since they need to have direct objects of their own.



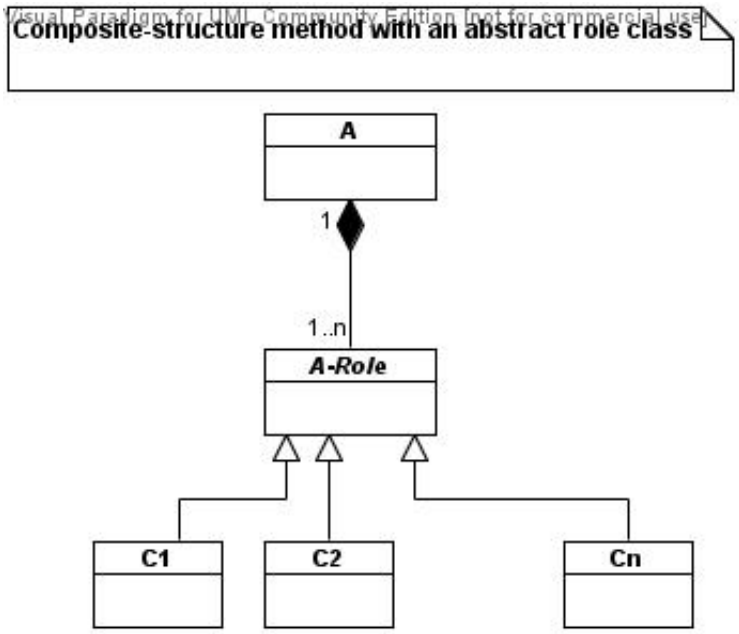


The composite-structure method, like multiple inheritance, eliminates the problems with the single-inheritance model described at the outset.

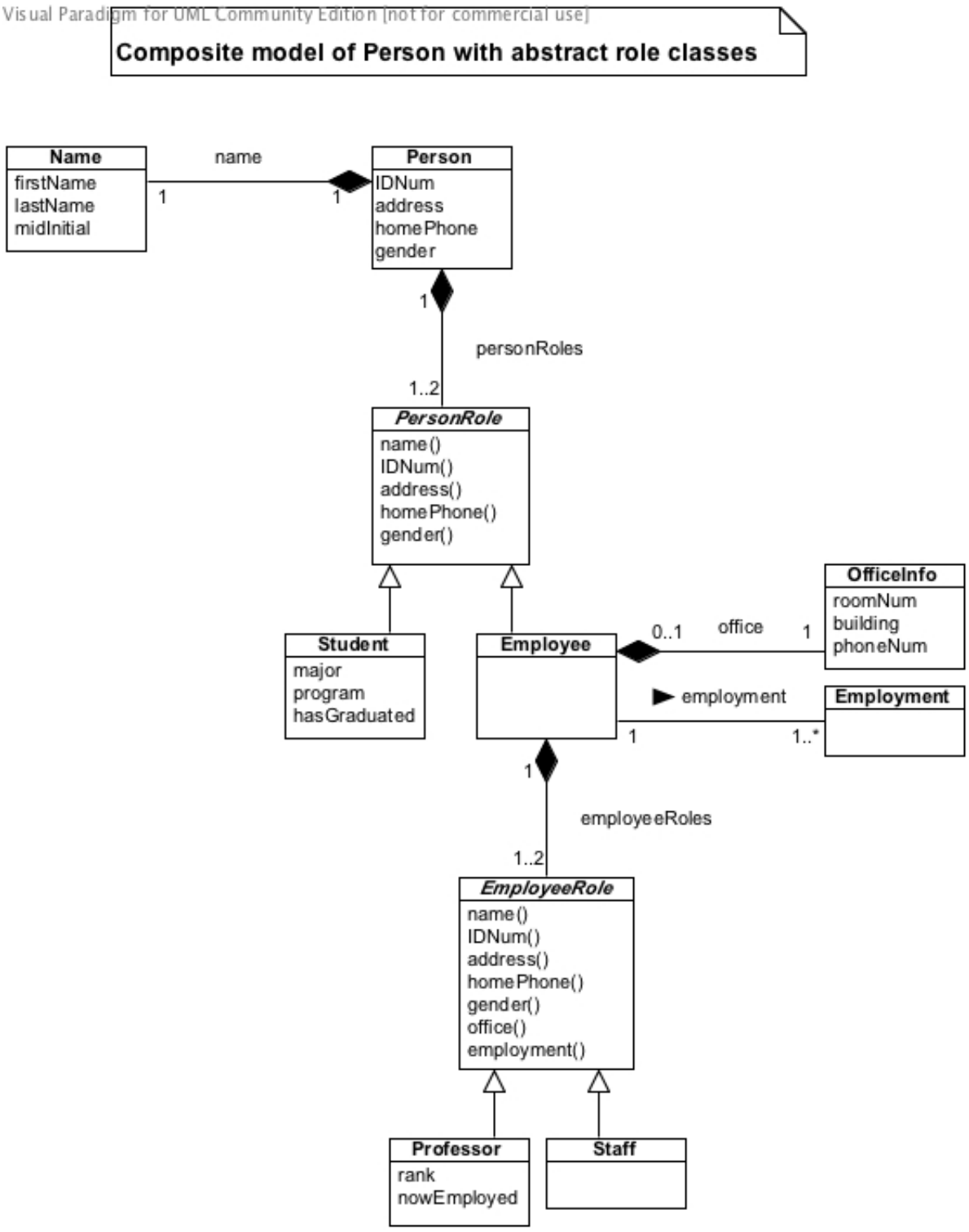
The composite-structure method can be used when the implementing database system does not support multiple inheritance. Even if it supports multiple inheritance, the method may be used when there is a danger of a combinatorial explosion of subclasses by multiple inheritance. In this method, however, the power of automatic inheritance of class features and inheritance polymorphism is lost.

One difficulty with this method concerns *emergent* attributes, functions, and relations that may appear in lower subclasses that would be derived by multiple inheritance. For example, suppose a certain numerical adjustment needs to be made to the salaries of professors who are also staff members. In the true multiple inheritance model, this can be naturally stored as an attribute or function of the ProfessorStaff class. The composite model has no natural place to store this feature. We may create a ProfessorStaff component class with the numerical adjustment feature and connect it to both Professor and Staff classes by weak composition relations. However, this would reintroduce a large number of such component classes – something the composite method is designed to eliminate. In general, if all of the $2^n - n - 1$ potential subclasses introduced emergent features, they would have to be stored somehow in the composite model, and combinatorial explosion seems unavoidable.

Use of General, Abstract Role Classes The composite-structure method can make use of general, abstract role classes from which specific role subclasses are derived.



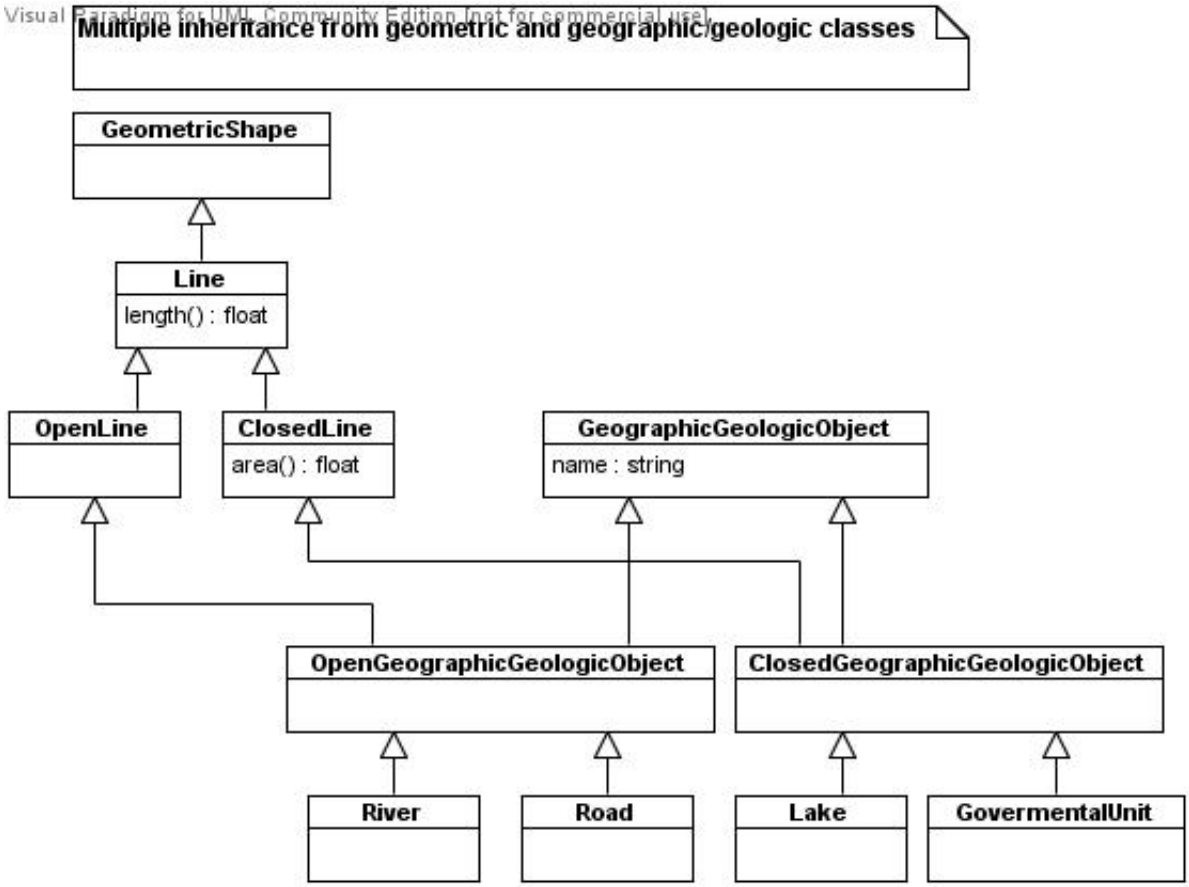
Here, the specific role classes C_i are derived from the general role class *A-Role* by inheritance. The class *A-Role* is abstract and contains suitable delegation functions to extract the class A's features. Compared to the previous method, this schema enhances generality and extensibility because any number of specific role classes can be derived from a single general role class. Note that inheritance is only used for specialization of the role class *A-Role*. Specialization of the class A itself is still represented by composites of A-objects with appropriate collections of C_i-objects. The following is an example of this method applied to the Person class.



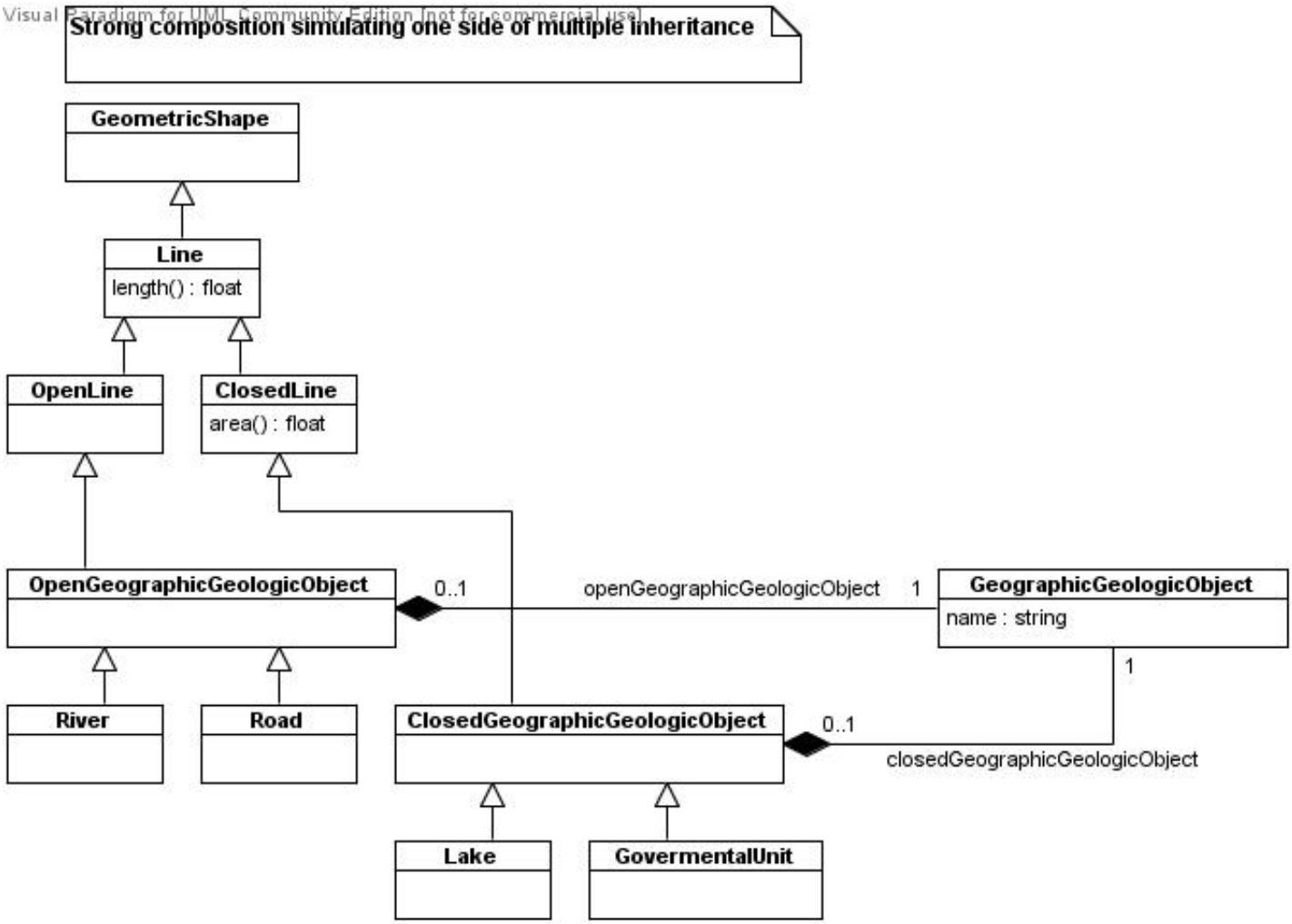
Consider a database for a digital library that stores the entire contents of academic books. Books in different subjects require different structures. Mathematics books, for example, require many symbols, formulas, and equations. Computer science books need to present algorithms and programs, and may include working software. Music books need to include scores and might also include digitally encoded music. We may use a hierarchical classification of books by multiple inheritance to incorporate structural variations. At the top of the hierarchy, we create the root class *Book* with common features like title, publisher, authors, ISBN, etc. We then derive subclasses in consideration of subject-specific data objects and structures. Multiple inheritance can be used to store interdisciplinary books dealing with multiple subjects such as computational mathematics, computational music, bioinformatics. This classification process can result in a large number of subclasses in a deep, dense multiple inheritance hierarchy as the number of interdisciplinary subjects increases. Therefore the composite-structure method described in this section might be a good alternative to a multiple inheritance hierarchy of books.

Absence of "Root" Class In the above discussion of the composite-structure method, we have assumed the existence of a class A that corresponds to the root class that serves as a common ancestor in an inheritance hierarchy. In some uses of multiple inheritance, such a class A may be absent or cannot be naturally identified. In this case a "dummy root class" A may be added, or an ad hoc composite-structure method would have to be used wisely.

Typical examples are application domains that use objects with spatial, geometric properties. Geographic/geologic information systems require representations of spatial properties of geographic/geologic objects such as open and closed lines on the earth's surface. Computer aided design/manufacturing systems require representations of 1-, 2-, and 3-dimensional objects such as lines, rectangles, rectangular volumes, and cylinders for machine components. A standard method for these kinds of applications is to develop a hierarchy of pure geometric classes and a hierarchy of physical-object classes separately, and then merge them appropriately by multiple inheritance. The following is a simplified model of geographic/geologic objects that have geometric shapes of open and closed lines.



Here the multiple inheritance is used to merge geometric shapes (open and closed lines) and geographic/geologic objects with other non-geometric characteristics which are, essentially, categorically disjoint from each other and themselves do not descend from a common ancestor. One option for simulating this multiple inheritance by the composite-structure method is to add a dummy root class from which the classes *GeometricShape* and *GeographicGeologicObject* are derived and then apply the method described previously. A second option is to use an ad hoc method to build a model like this:



Both *OpenGeographicGeologicObject* and *ClosedGeographicGeologicObject* should include delegation functions to traverse the strong composition relations to extract the features of *GeographicGeologicObject*. Alternatively, *OpenGeographicGeologicObject* and *ClosedGeographicGeologicObject* could be subclasses of *GeographicGeologicObject* with strong composition relations to *OpenLine* and *ClosedLine*, respectively.