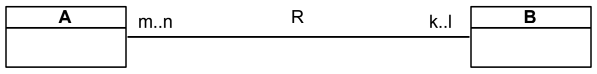
CS 332/780: Object-Oriented Databases   Spring 2018
Course Notes #5
Keitaro Yukawa
Department of Computer Science
Queens College, CUNY

Mapping UML Class Diagrams to Class Schemas

Although not a modeling issue, it will be beneficial at this time to discuss a general method for mapping UML class diagrams to class schemas in implementation OODB systems. Mapping of class attributes is straightforward: simply implement them by choosing appropriate data types available in the implementation OODB system. Thus we only need to discuss mapping of binary relations: association, composition, and aggregation.

We assume that a set or collection type is provided in the implementation OODB system. Here we use the notation Set<T> where T is any basic or class type available in the system. If T is a class type, any value of Set<T> type will be a set of *references* to objects of class T.

Consider any association, composition, or aggregation relation R(A, B):



If l = 1: In class A, create an attribute R of type B:

```
  class A
  {  ...
     ...
     B  R;
  }
```

If l > 1 or l = *: In class A, create an attribute R of type Set<B>:

```
  class A
  {  ...
     ...
     Set<B>  R;  // holds R↓B(a) = { b ∈ B | R(a, b) } where a is "this" A-object.
  }
```

In place of Set<B>, types like List<B> or OrderedSet<B> may be used if it is provided in the implementation OODB.

The inverse relation R⁻ can be implemented in two ways. If frequent traversals from B to A are expected or traversal efficiency is critical, implement R⁻ by an attribute in class B to hold R⁻↓A(b) = { a ∈ A | R⁻(b, a) } using the above method. Otherwise, implement a method function for R⁻ in class B to compute R⁻↓A(b) dynamically:

If n = 1:

```
  A  RInverse()
  {
     find the A-object that relates by R to the target B-object and return it
  }
```

If n > 1 or n = *:

```
  Set<A>  RInverse()
  {
     find the set of A-objects that relate by R to the target B-object and return it
  }
```

These functions can be computed by using the attribute R in the class A, either by the query language or by algorithmic means. Computing these functions can be inefficient if the extent of class A is large, so this alternative should only be used if infrequent traversals from B to A are expected or efficiency is not critical. On the other hand, it saves space that would be needed for an explicit attribute in class B implementing R⁻, an instance of space-time trade-off. It also saves the need to maintain consistency between the attribute R in A and the attribute R⁻ in B.

If a relation R(A, A) is symmetric then R = R⁻, so the inverse R⁻ need not be implemented.

The following example class schemas implement *Department* and *Course* classes in the college model at the end of Course Notes #4.

```
class Department extends Office
{
        Professor chairPerson;  // Implements chairPerson: Department → Professor  0..1, 0..1
        Set<Course> offer;  // Implements offer: Department → Course  1..*, *
}

class Course
{
        String number;
        String title;
        int credits;
        int hours;
        String description;

        Set<CourseSection> courseSection; // Implements courseSection: Course → CourseSection  1, *
        Set<Department> offeredBy(); // Returns the set of departments that offer this course.
                                     // Implements the inverse offer¯: Course → Department  *, 1..*
}
```

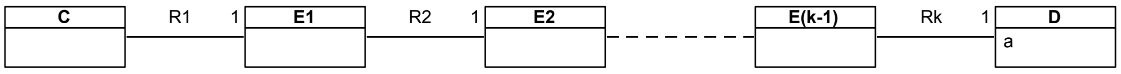
Keys

A set of attributes { A_1, ..., A_n } of a class C is a *key* for C if the following two conditions are satisfied:

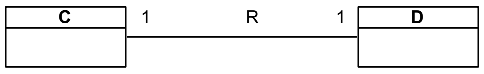
1. Each C-object has non-null values for all of A_1, ..., A_n at all times;
2. No two C-objects have the same values for all of A_1, ..., A_n at any time.

The condition 2 implies that at most one C-object has any given set of key attribute values { a_1, ..., a_n }. This, combined with the condition 1, ensures that every C-object is uniquely identified by its key. The C-objects and their key values are in 1-to-1 correspondence. Thus a key is frequently used for retrieving single objects. A key usually consists of a single attribute, like student ID or bank account number. A key may contain two or more attributes. For example, in some colleges registration codes uniquely identify course sections only within one semester, but three attributes { registration code, semester, year } uniquely identify all course sections across different semesters and years.

In the above definition it is useful to extend the meaning of attributes to include those that can be reached from the class C by relation compositions $R_1R_2 \cdots R_k$: C → D where each R_i is a binary relation with the multiplicity of exactly one on the target class's side. For every C-object *c*, exactly one D-object is reachable from *c* by the composition $R_1R_2 \cdots R_k$. This object is $(R_1R_2 \cdots R_k)$↓D(c), abbreviated to c.$(R_1R_2 \cdots R_k)$. This D-object and any attribute *a* of the class D, *c*.$(R_1R_2 \cdots R_k)$.*a*, are regarded as extended attributes of C, and will be denoted simply by $R_1R_2 \cdots R_k$ and $(R_1R_2 \cdots R_k)$.*a*, respectively. A key for C may then include extended attributes.

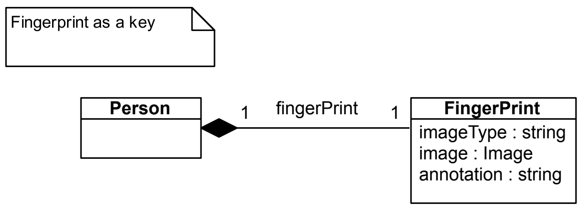


Consider the example college model in Course Notes #4. Since every Employee object has exactly one OfficeInfo object and no OfficeInfo object is shared by more than one Employee object, the strong composition relation *office* is a key for Employee. On the assumption that the Professor class only contains full-time professors who are each assigned a unique office room with a phone number, either of office.phoneNum and {office.roomNum, office.building} uniquely identifies a professor's room, so they too form keys for Professor. Because every pair of a Student object and a CourseSection object is associated with at most one Take object, {student, courseSection} is a key for Take. Hence the union of a key for Student and a key for CourseSection forms a key for Take, for example {student.IDNum, courseSection.registrationCode, courseSection.semester, courseSection.year}. For any binary relation R(C, D) with the multiplicity of 1 on both sides, R is a key for C and R¯ is a key for D, that is, each D-object uniquely identifies the related C-object and vice versa.



In the example college model, this is the case with grade(Take, Grade), name(Person, Name), and jobDescription(Employment, JobDescription). Each Grade-, Name-, or JobDescription-*object* is a key for Take, Person, or Employment, respectively. However, no attribute set of Grade, Name, or JobDescription forms a key for the respective class, hence it does not serve as a key for Take, Person, or Employment either.

A key need not be a primitive value and can be a complex object. For example, biometrics such as fingerprints, eye prints, and DNA profiles have been increasingly used as keys for persons. Extended attributes are useful for forming these types of complex keys. Below the FingerPrint class is a strong composition component of Person:



Here either of fingerPrint and fingerPrint.image can be designated as a key for Person.

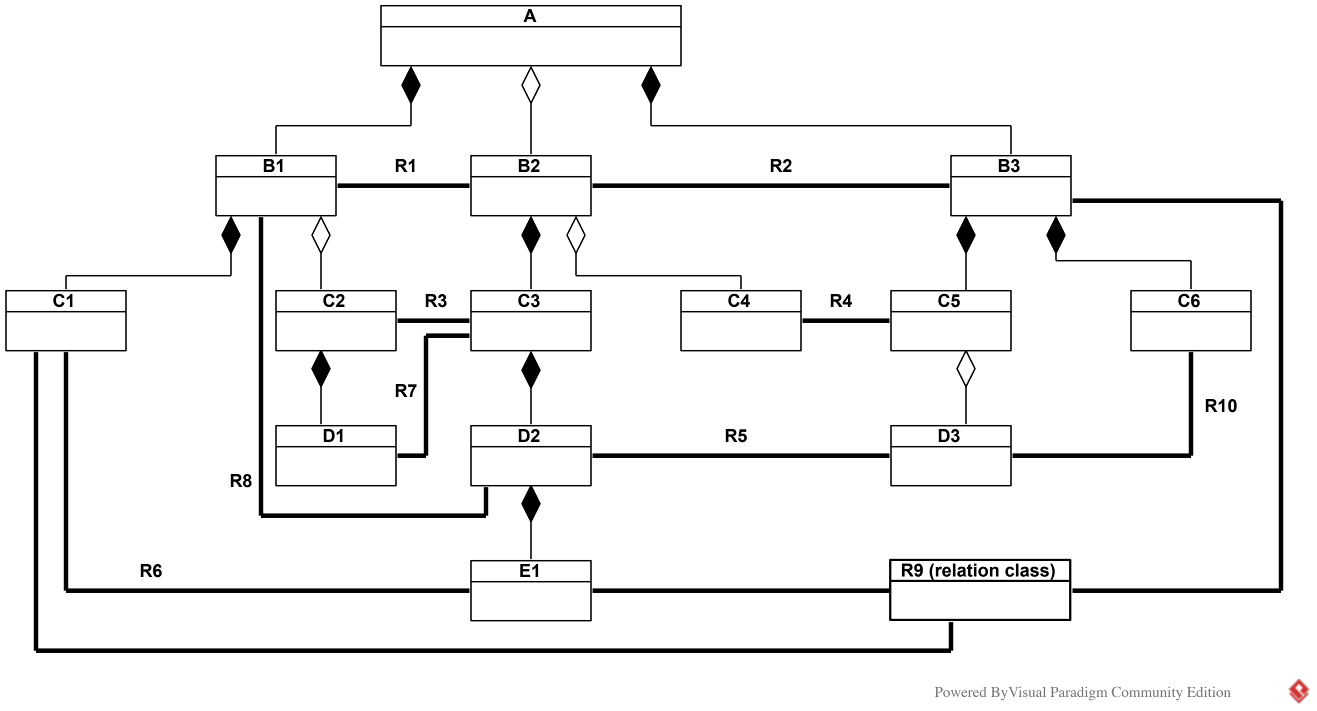
Biometric data should be used with caution for the obvious reason of security and privacy. Biometric signatures, if stolen, cannot be changed (at least until bio-genetic science/engineering is advanced to a point when they can be altered).

In OODBs (and in object-oriented languages generally), every object is uniquely identified by a reference pointer to it and also by a system-generated object identifier. Some systems combine the two by using (symbolic codes of) reference pointers as system-generated object identifiers. Thus classes in OODBs need not have key attributes. The Name, Grade, OfficeInfo, JobDescription, Take, and Employment classes in the college model do not have keys (of non-extended attributes). For example, multiple Name objects may have the same first name, last name, and middle initial – these are distinguished by their object identifiers and/or reference pointers to them. This is one reason why OODBs are said to be "reference oriented" as opposed to "value oriented". The role of keys in OODBs is therefore not as crucial as in relational DBs. In fact UML does not provide any special notation for keys; key attributes of a class, if any, may be specified in a text documentation area.

The concept of key is a form of logical constraint and should be distinguished from *indexes*, which are data structures for efficient retrieval of information. Since a key uniquely identifies a single object, it is natural to create an index on it to make object retrieval by the key attributes efficient. (Due to this circumstance "keys" and "indexes" have been sometimes used interchangeably, and confusingly. In the context of databases, it is best to heed the distinction carefully, and we shall do so in this course.) For example, retrieval of students or bank accounts by student ID or bank account number is very frequent so it is advantageous to create an index on these attributes for superior efficiency. However, indexes can also be created on non-key attributes. Later in the course we will study two common data structures used for indexes: B$^+$ trees and hash tables.

Hierarchical Decomposition of Aggregate/Composite Objects into Constituent Parts and Inter-Part Local Relations

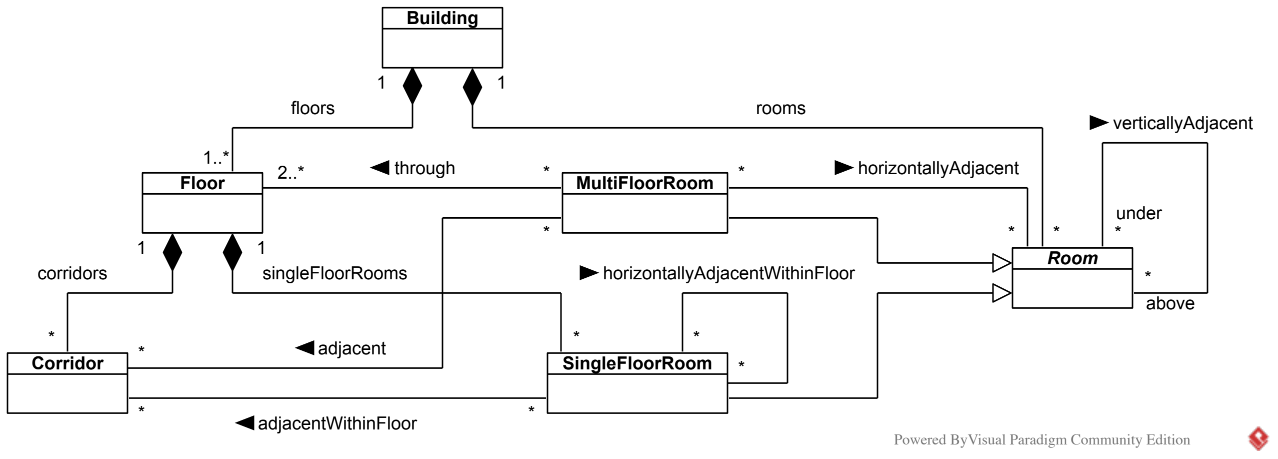
Complex aggregate/composite objects consist of layers of parts together with internal relations representing their configurations, i.e., the manners of their functional or physical interconnections. These inter-part relations are different in character from whole-part relations because they are association relations among part objects. A significant property of inter-part association relations is that they are locally confined to a single aggregate or composite whole and do not extend outside of it; in this sense they are said to be *local* to the aggregate or composite whole. In modeling a database of computer architectures, we need to establish various association relations among crucial components like processors, main memories, hard disks, graphics cards, etc. The locality of these relations consists in the fact that they never relate components of different computer models exclusively, in the sense that if components are related, there must be a whole computer model of which they are all parts. When an aggregate or composite forms a hierarchy of parts, subparts, sub-subparts, etc., inter-part relations may exist among parts at the same level or different levels. In databases to support computer-aided design/manufacturing systems for major engineering artifacts like ships, aircraft, and spacecraft, for example, thousands of components, down to nuts and bolts, have to be organized by dozens of whole-part layers and hundreds of inter-part relations.



A single class diagram of an aggregate/composite fails to model locality of inter-part relations, and may become too dense to exhibit the layered structure of parts clearly. At any rate it is impractical to construct a single class diagram for all the components and relations in realistically complex objects like airplanes, ships, and buildings. *Hierarchical decomposition* of aggregate/composite objects is an effective, modular way to model the detailed structure of parts and inter-part local relations in layer-by-layer manner.

Man-made machines and devices are not the only examples of complex objects. Other examples include galaxies, geological entities, biological organisms, chemical compounds, molecules, atoms, companies, governmental offices/organizations, universities, documents like websites, magazines, newspapers, tax forms, etc. Indeed, real-world application domains abound with complex objects comprised of whole-part structures and inter-related parts.

For example, consider the following model of buildings.



The following relations are confined to a single building, but are not confined to a single floor in that the relations and/or the participating objects may cross floors:

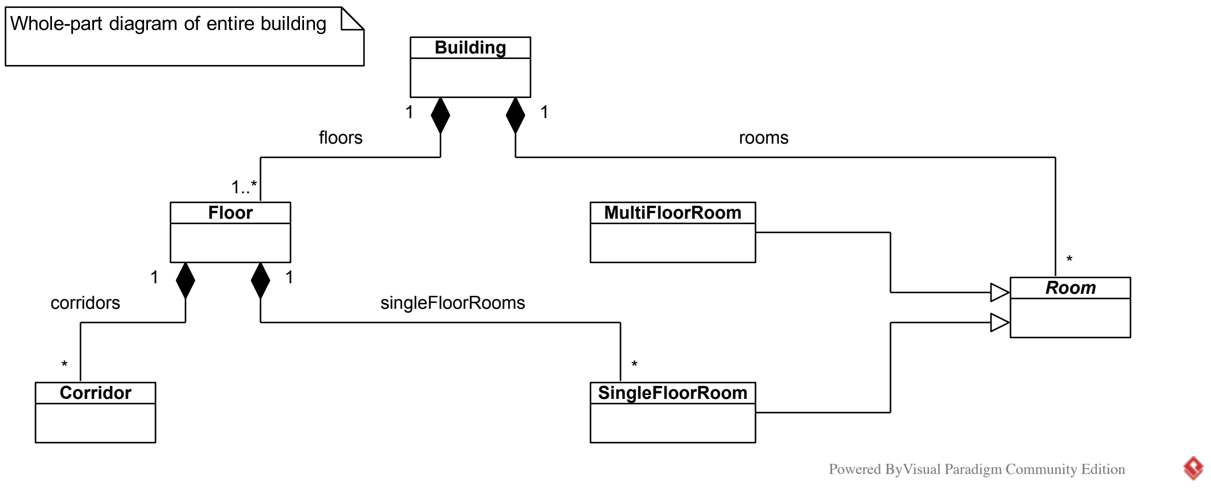
verticallyAdjacent(Room, Room)
horizontallyAdjacent(MultiFloorRoom, Room)
adjacent(MultiFloorRoom, Corridor)
through(MultiFloorRoom, Floor)

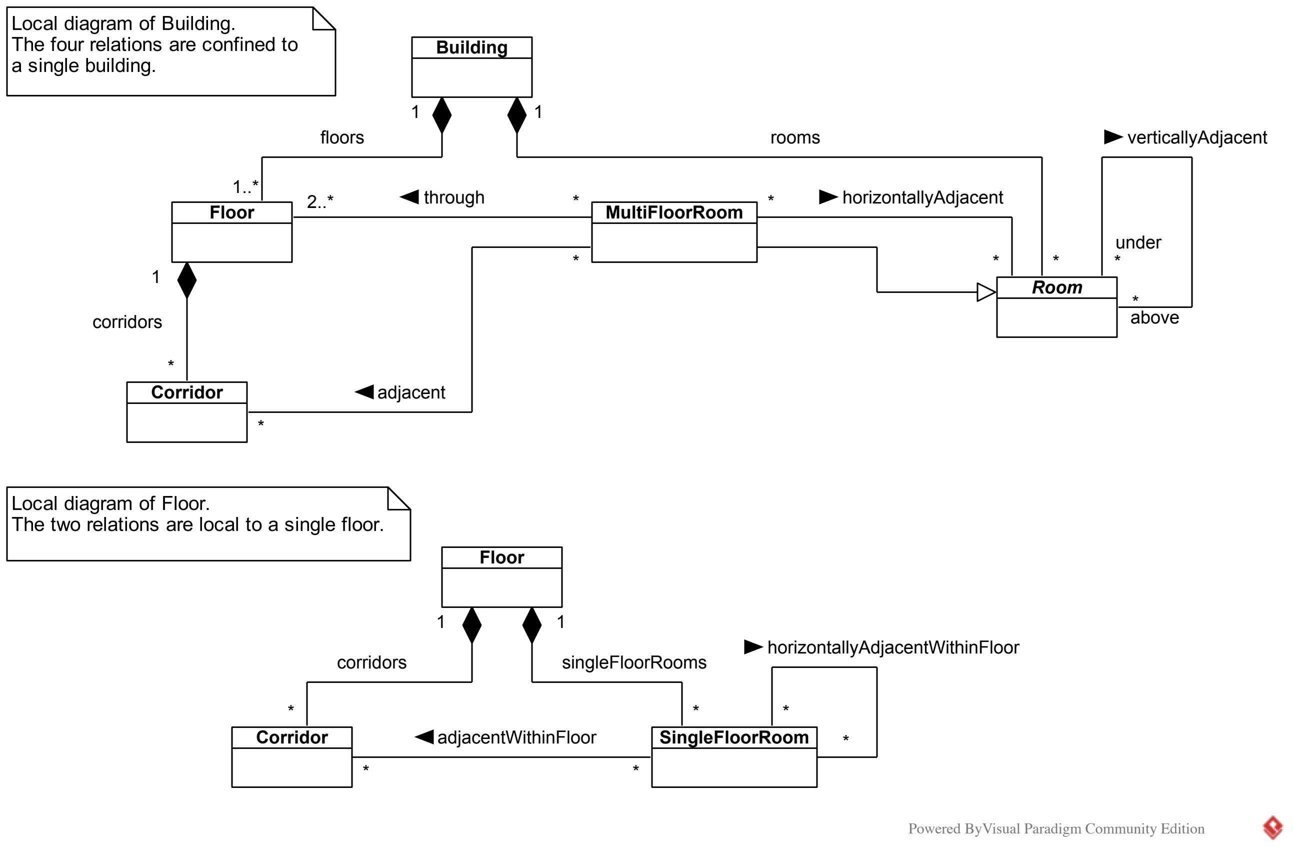
The following relations are confined to a single floor:

horizontallyAdjacentWithinFloor(SingleFloorRoom, SingleFloorRoom)
adjacentWithinFloor(SingleFloorRoom, Corridor)

The above "one-for-all" diagram fails to impose these locality constraints. For example, it allows two rooms in different buildings to be related by *verticallyAdjacent*, and allows two single-floor rooms on different floors to be related by *horizontallyAdjacentWithinFloor*. Although these locality constraints can be specified in a text documentation area, it is more comprehensible and informative to use a hierarchical collection of *local class diagrams* to display inter-part relations and their local scopes. To this end, the above building diagram is decomposed into three class diagrams:

- **whole-part diagram**: displays only the whole-part and inheritance relations of the entire building – no inter-part association relations are shown;
- **local diagram of Building**: displays the direct parts of a building and the four inter-part relations confined to a single building but not confined to a floor or a room;
- **local diagram of Floor**: displays the direct parts of a floor and the two inter-part relations confined to a single floor but not confined to a corridor or a single-floor room.





In the local class diagram of Building, the Corridor class appears due to its being an argument of the *adjacent* relation although it is not a direct part of Building.

In practice, locality constraints can be ensured by performing suitable constraint checking whenever a new relation instance is created. For example, whenever two rooms are entered into the *verticallyAdjacent* relation, the responsible function would make sure that they belong to the same building.

We now describe a general decomposition method, beginning with a general definition of local relations.

Assume we have a hierarchy of whole-part relations for a certain class of composite/aggregate objects. A B-object *b* is a *part* of an A-object *a* in this hierarchy if there is a chain of composition/aggregation relations ($R_1 \cdots R_k$): A → B, k ≥ 1, such that ($R_1 \cdots R_k$)(*a*, *b*) holds. An inter-part association relation R(C_1, ..., C_n) is *local* to a class C in this hierarchy if for each relation instance R(x_1, ..., x_n), there is a C-object of which all the x_j are parts in the above sense. If at least one of n whole-part relations from C to C_i, 1 ≤ i ≤ n, is strong composition, then there cannot be distinct C-objects of which the x_j are parts, for it would violate the uniqueness of the whole requirement of strong composition, hence there is a *unique* C-object of which all the x_j are parts. If all of n whole-part relations from C to C_i are weak composition, there may be distinct C-objects of which the x_j are parts.

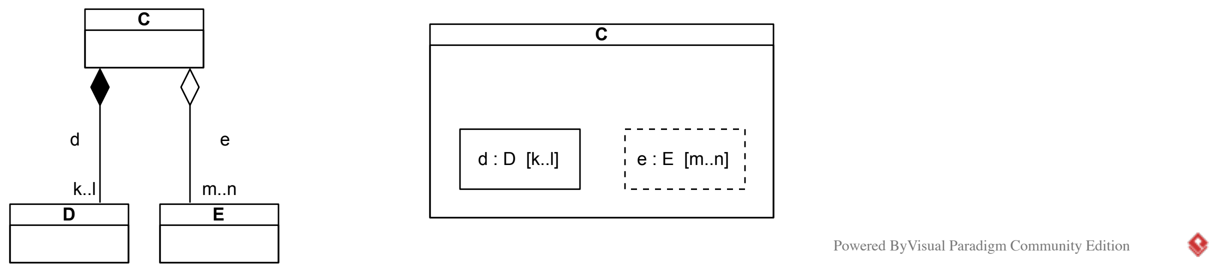
The following is a hierarchical decomposition method based on the above definition of local relations.

- Provide a **whole-part diagram** displaying the overall hierarchy of whole-part relations, possibly along with inheritance relations. This diagram contains no inter-part association relations.
- Provide a **local class diagram** for each class C which appears in the whole-part diagram and has at least one part class. This diagram contains the inter-part relations local to C but not local to any of C's part classes, along with the relations' argument classes. Also include the whole-part and inheritance relations connecting these classes.
- In case there are inter-part relations not local to the root class, provide a **nonlocal relation diagram** wherein the nonlocal inter-part relations are overlaid on the whole-part diagram.

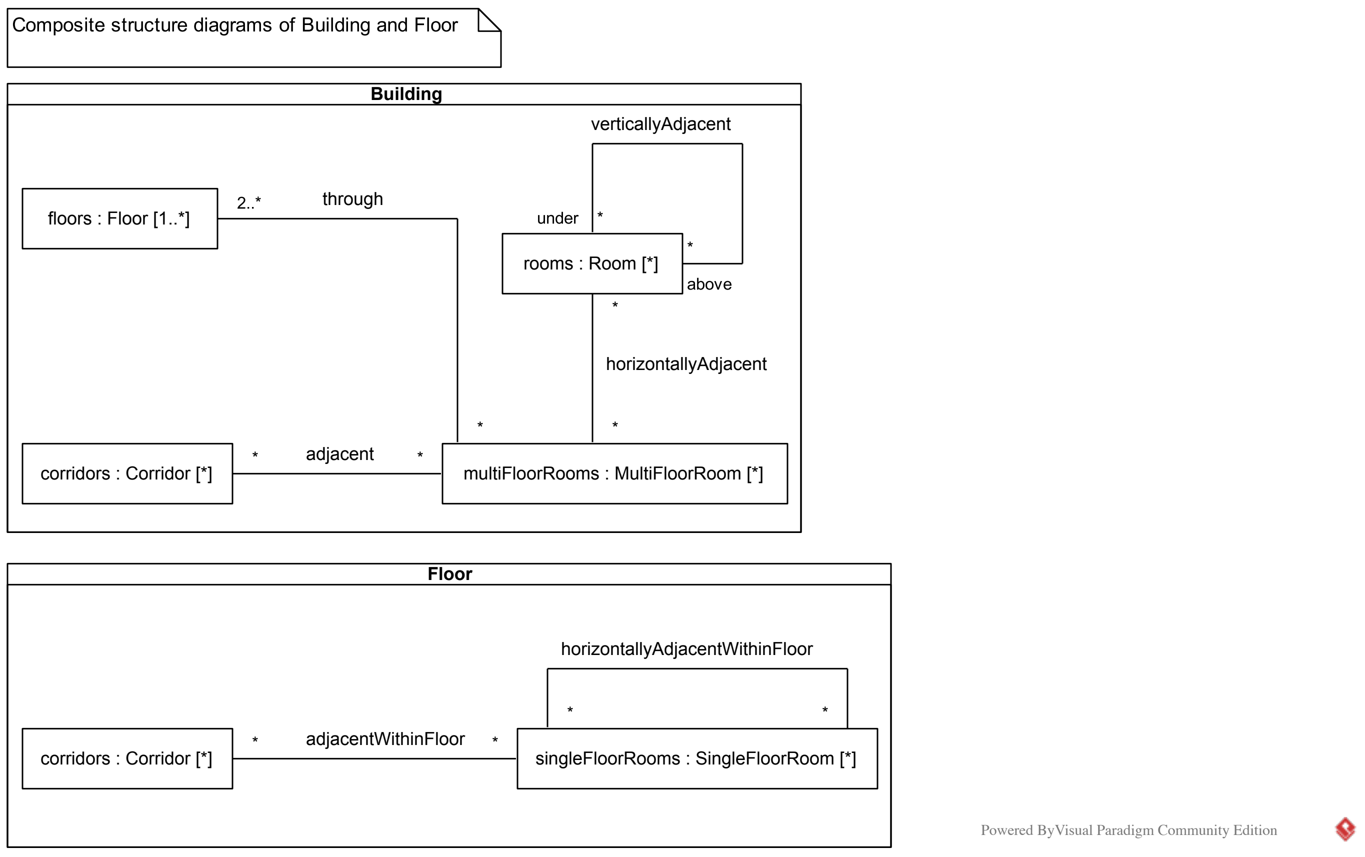
An example of nonlocal relation in the building model is passageway(Corridor, Corridor) where passageway(c_1, c_2) means there is a passageway (e.g., bridge, tunnel) between corridors c_1 and c_2 in different buildings.

It should be pointed out that since local class diagrams are normal class diagrams and not a special feature of UML, they should be properly annotated to document that the displayed relations are local to the pertinent classes.

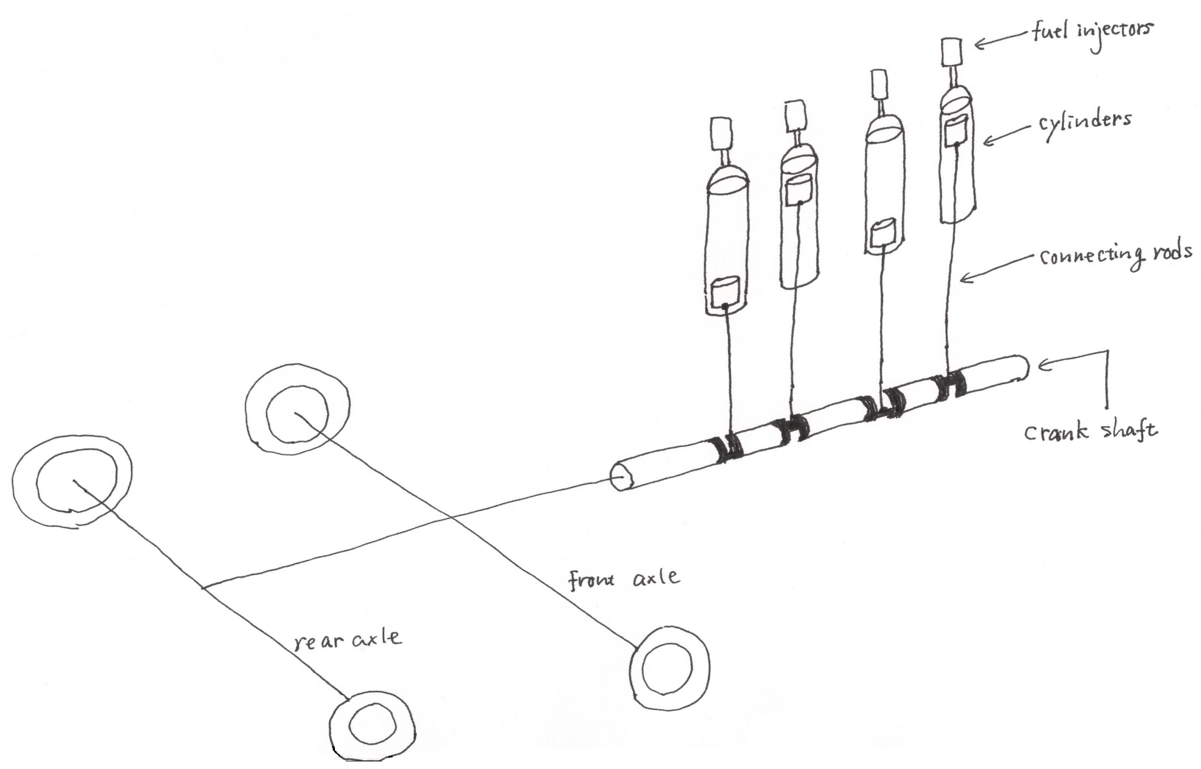
Instead of local class diagrams, we may use *composite structure diagrams* introduced in UML version 2.0. A class C is displayed by a box containing its parts and inter-part local relations. Part boxes with solid outlines represent strong-composition parts, while those with dashed outlines represent aggregation parts. The local class diagram on the left below is equivalent to the composite structure diagram on the right.

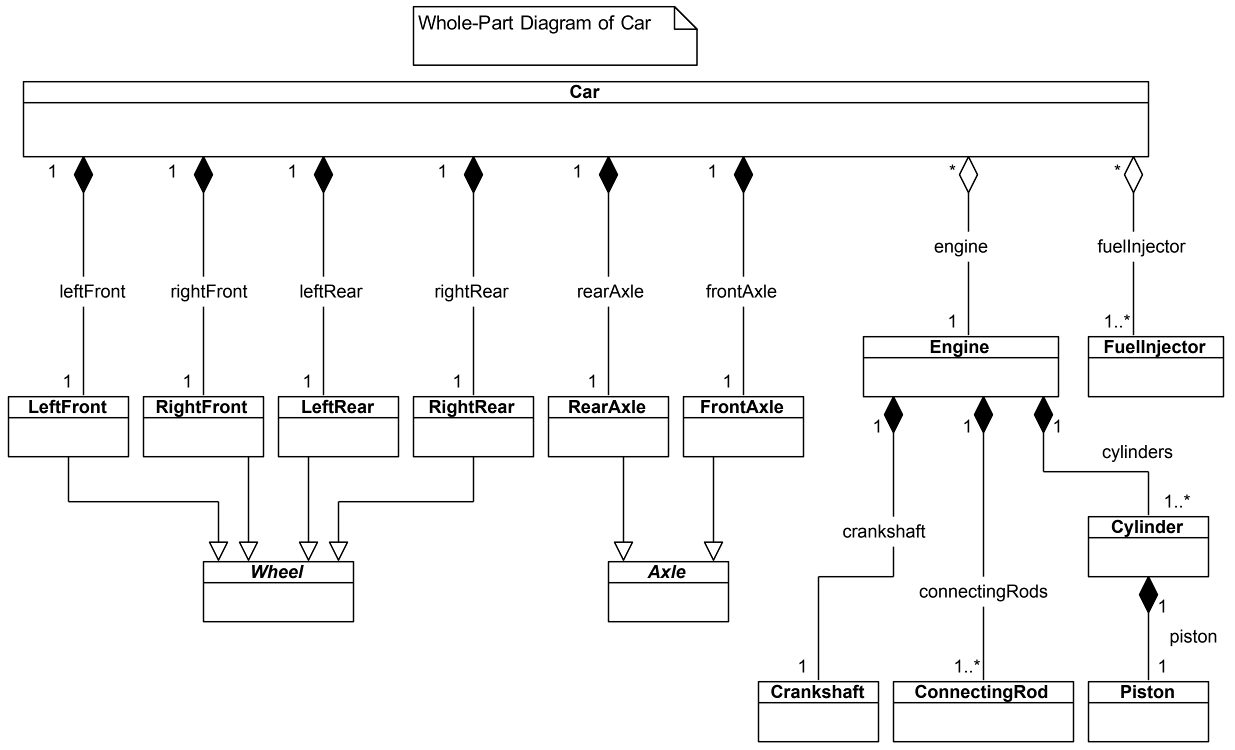


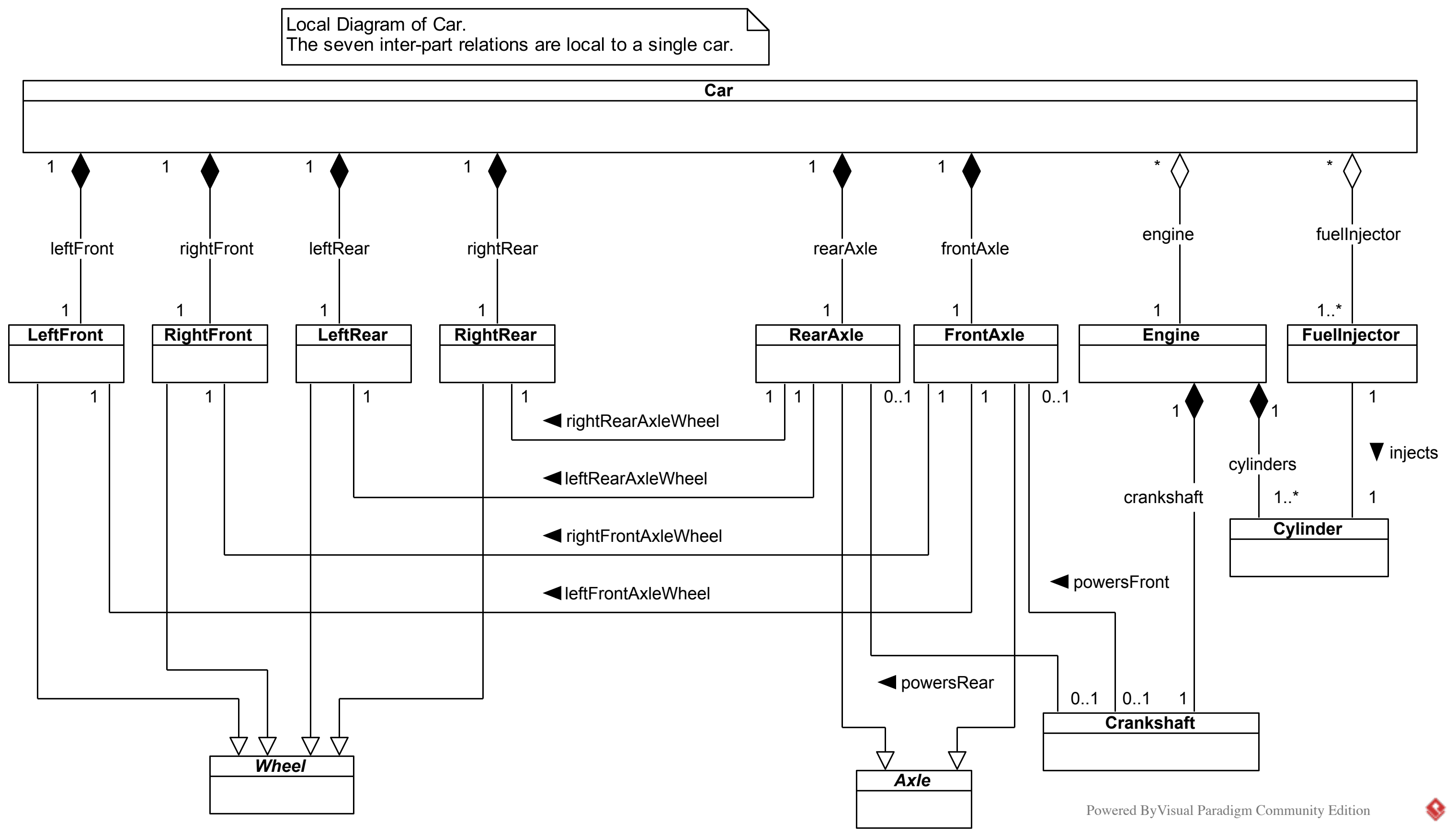
In the C box on the right, *d*, *e* are called part names and *D*, *E* are called part types. So the use of composite structure diagrams in hierarchical decomposition amounts to the use of box notation in place of composition/aggregation relation lines. Unlike local class diagrams, however, the locality of inter-part relations in composite structure diagrams is a *built-in requirement*. So all the relations displayed in a class box are mandated to be local to individual objects of the class, in need of no annotations to this effect. The following composite structure diagrams correspond to the local diagrams for Building and Floor.

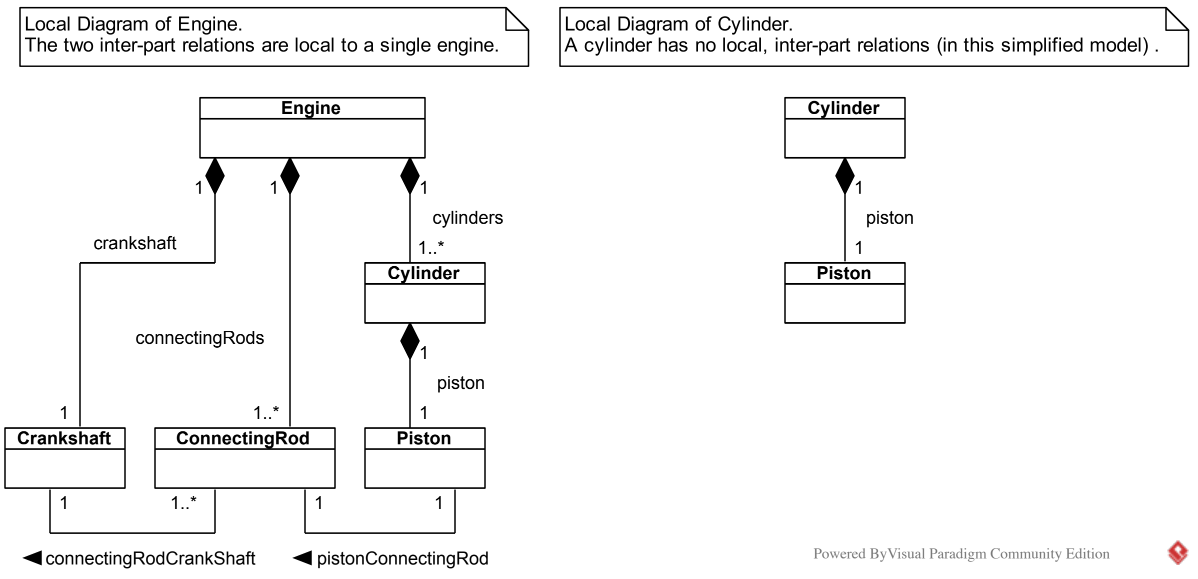

Composite structure diagrams of Building and Floor
Building
verticallyAdjacent
floors : Floor [1..*]
2..*
through
under
*
rooms : Room [*]
*
above
*
horizontallyAdjacent
*
*
corridors : Corridor [*]
*
adjacent
*
multiFloorRooms : MultiFloorRoom [*]
Floor
horizontallyAdjacentWithinFloor
*
*
corridors : Corridor [*]
*
adjacentWithinFloor
*
singleFloorRooms : SingleFloorRoom [*]
Powered ByVisual Paradigm Community Edition

In the following car model, aggregation relations are used for the engine and fuel injectors, while strong composition relations are used for all else. Each Car-object represents a car model. Each Engine-object represents an engine model – likewise for the other component classes. Hierarchical decomposition is modeled by both local class diagrams and composite structure diagrams.


fuel injectors
cylinders
connecting rods
crank shaft
front axle
rear axle


Whole-Part Diagram of Car
Car
1
leftFront
1
LeftFront
1
rightFront
1
RightFront
1
leftRear
1
LeftRear
1
rightRear
1
RightRear
1
rearAxle
1
RearAxle
1
frontAxle
1
FrontAxle
*
engine
1
Engine
*
fuelInjector
1..*
FuelInjector
Wheel
Axle
1
crankshaft
1
Crankshaft
1
connectingRods
1..*
ConnectingRod
1
cylinders
1..*
Cylinder
1
piston
1
Piston

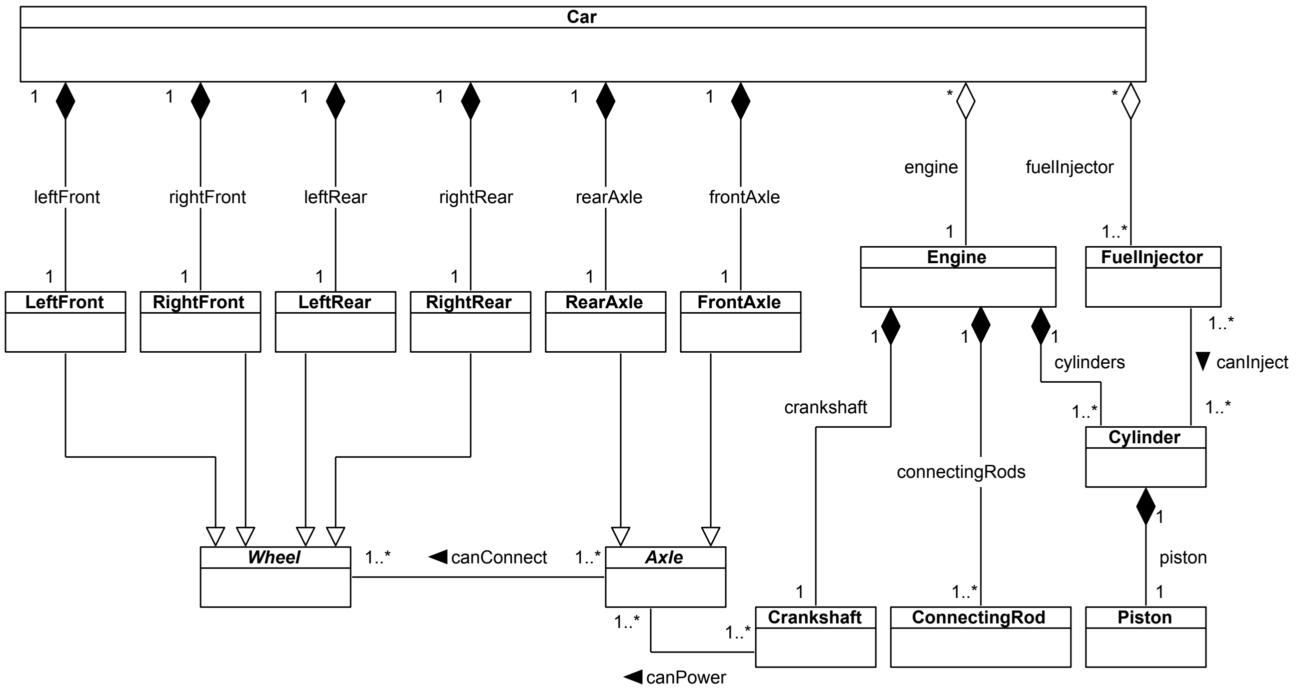

Local Diagram of Car.
The seven inter-part relations are local to a single car.
Car
1
leftFront
1
1
rightFront
1
1
leftRear
1
1
rightRear
1
1
rearAxle
1
1
frontAxle
1
*
engine
1
*
fuelInjector
1..*
LeftFront
RightFront
LeftRear
RightRear
RearAxle
FrontAxle
Engine
FuelInjector
◀ rightRearAxleWheel
◀ leftRearAxleWheel
◀ rightFrontAxleWheel
◀ leftFrontAxleWheel
0..1
0..1
◀ powersFront
◀ powersRear
crankshaft
cylinders
1..*
▼ injects
Cylinder
0..1
0..1
Crankshaft
Wheel
Axle

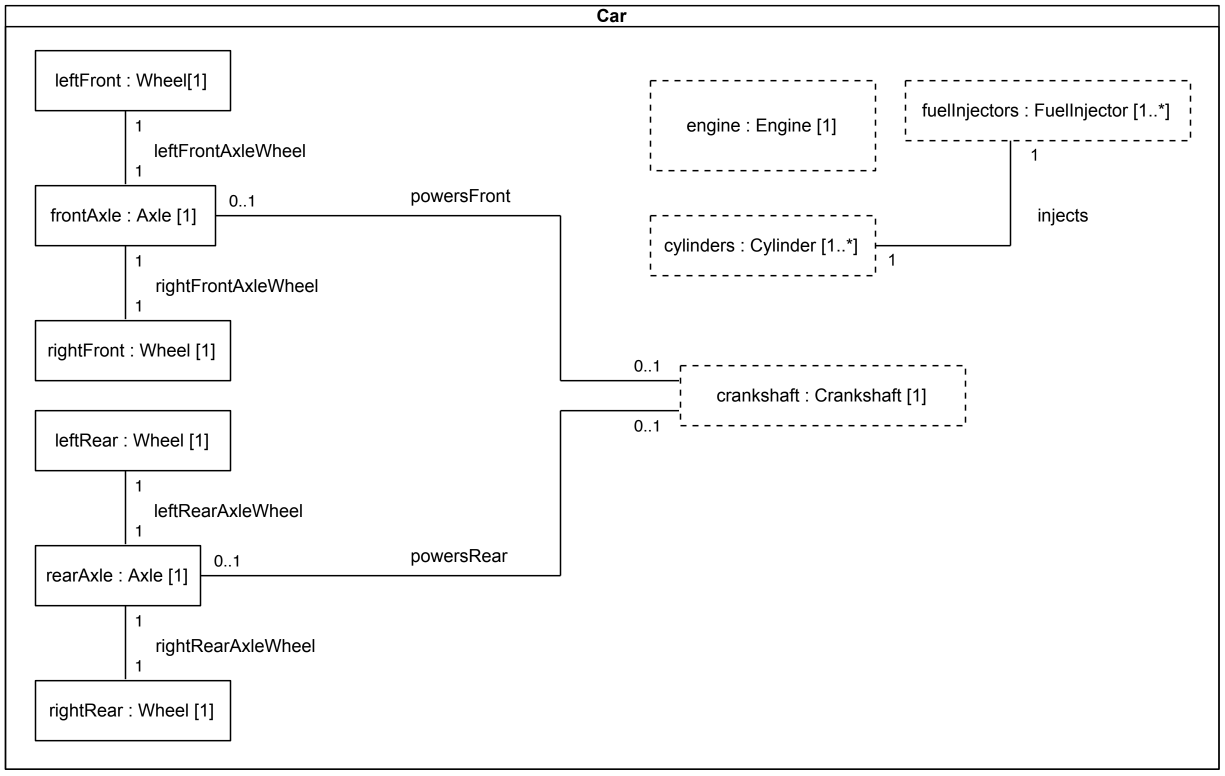

Local Diagram of Engine.
The two inter-part relations are local to a single engine.
Engine
1
1
1
crankshaft
connectingRods
cylinders
1..*
Cylinder
1
piston
1
1
1..*
Crankshaft
ConnectingRod
Piston
1
1..*
1
1
◀ connectingRodCrankShaft
◀ pistonConnectingRod
Local Diagram of Cylinder.
A cylinder has no local, inter-part relations (in this simplified model) .
Cylinder
1
piston
1
Piston

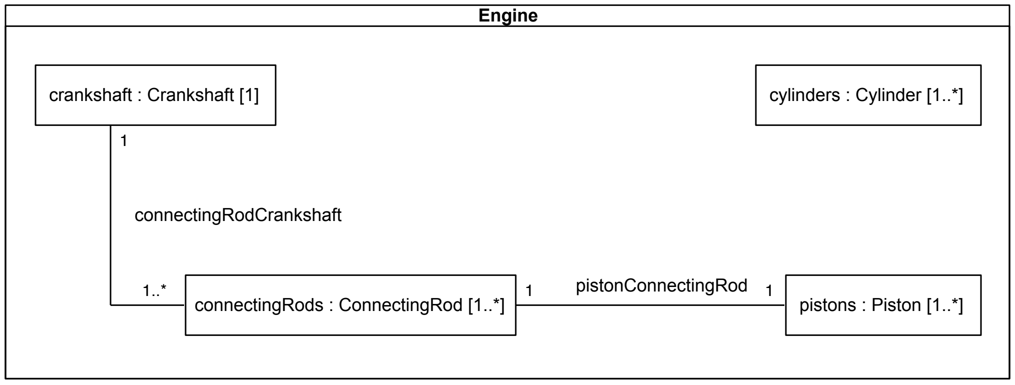
Nonlocal Relation Diagram

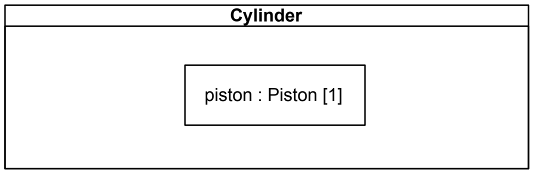
The following relations are not local to a single car model:

canInject(FuelInjector, Cylinder)
canPower(Crankshaft, Axle)
canConnect(Axle, Wheel)




Car
leftFront : Wheel[1]
1
leftFrontAxleWheel
1
frontAxle : Axle [1]
0..1
powersFront
1
rightFrontAxleWheel
1
rightFront : Wheel [1]
leftRear : Wheel [1]
1
leftRearAxleWheel
1
rearAxle : Axle [1]
0..1
powersRear
1
rightRearAxleWheel
1
rightRear : Wheel [1]
engine : Engine [1]
fuelInjectors : FuelInjector [1..*]
1
injects
cylinders : Cylinder [1..*]
1
0..1
crankshaft : Crankshaft [1]
0..1


Engine
crankshaft : Crankshaft [1]
cylinders : Cylinder [1..*]
1
connectingRodCrankshaft
1..*
connectingRods : ConnectingRod [1..*]
1
pistonConnectingRod
1
pistons : Piston [1..*]


Cylinder
piston : Piston [1]