Relationship between objects

Types of relationships

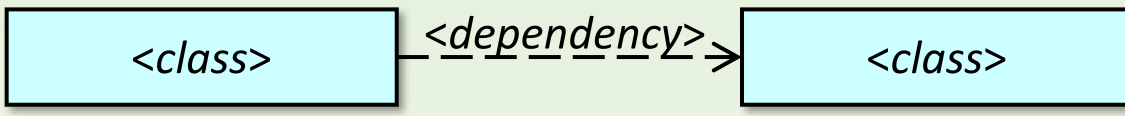
- When objects communicate with each other (they call the methods of each other synchronously or asynchronously, they send signals to each other, or they operate with the attributes of the other), then they establish relationship between each other.
- There are five types of relationships between objects:
  - dependency
  - association
  - aggregation or shared aggregation
  - composition or composite aggregation
  - inheritance

Abstraction

- *Objects* and their relationships (*link*) are represented by object diagrams, but objects and the properties of their relationship are shown in a class diagram, on a higher abstraction level.
- Class diagram is the abstraction of the object diagram.
- Unfortunately, in most cases, the programming languages do not provide tools to describe the relationships between objects (except the inheritance), they only know the abstraction of objects, which are the classes.

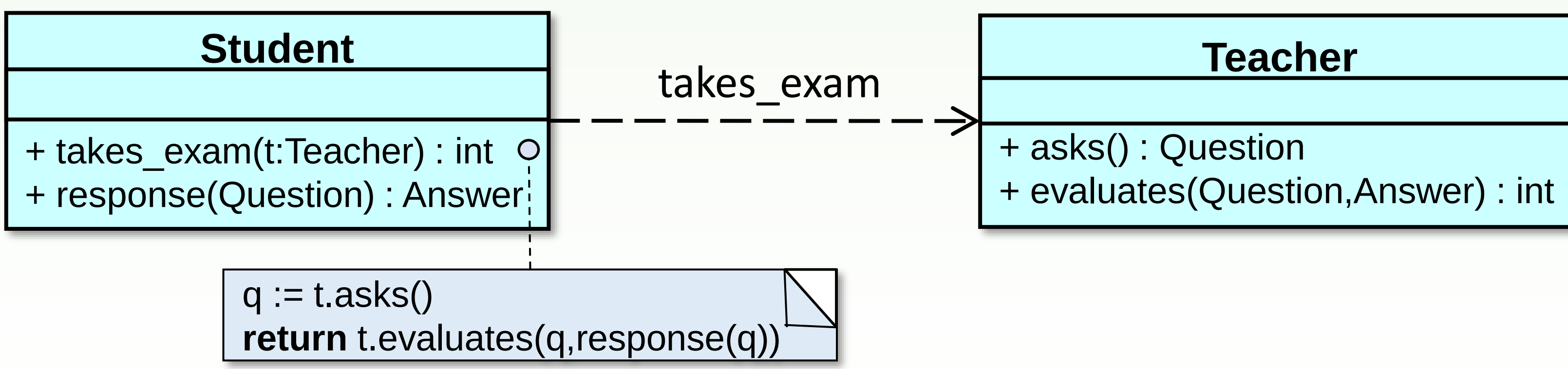
4 Important criterion of object orientation is abstraction.

Dependency



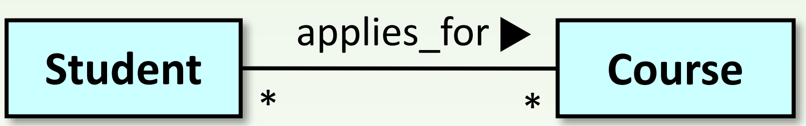
- When a method of a class gets in touch with an object of another class temporary, e.g. gets it as a parameter or locally instantiates it to call its method, to send a signal to it, or to forward the reference of the object (for example at exception throwing).

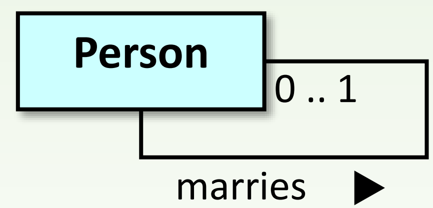
- When a method of a class calls a class-level method of an other class.



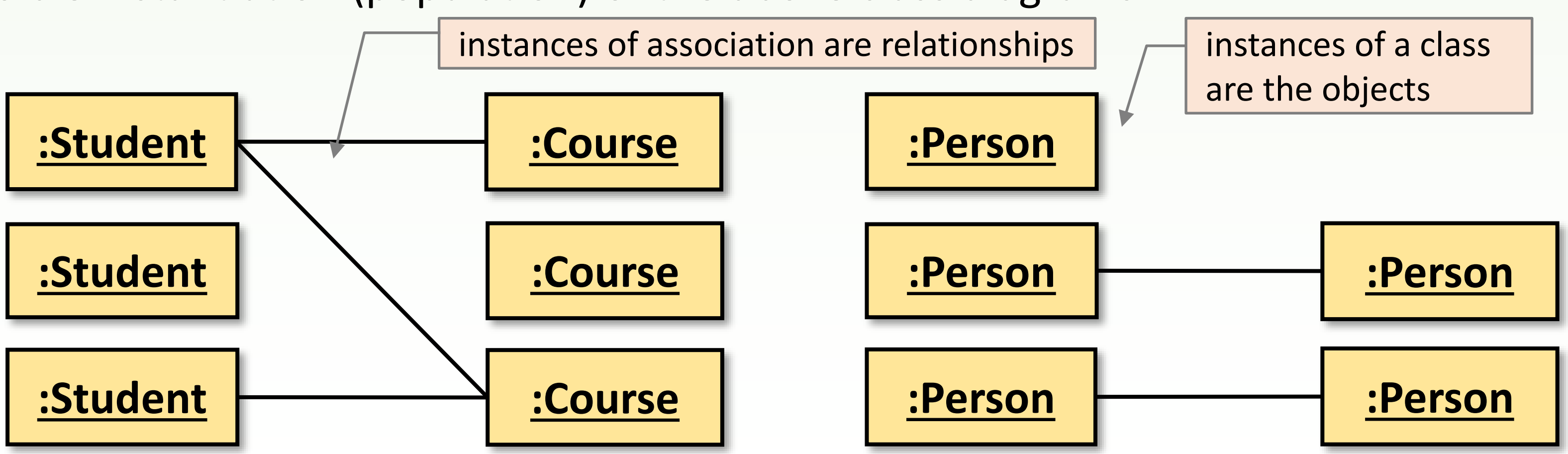
Association

- Expresses a long-term relationship between objects (permanent dependency between objects).
- One association might describe several relationships.



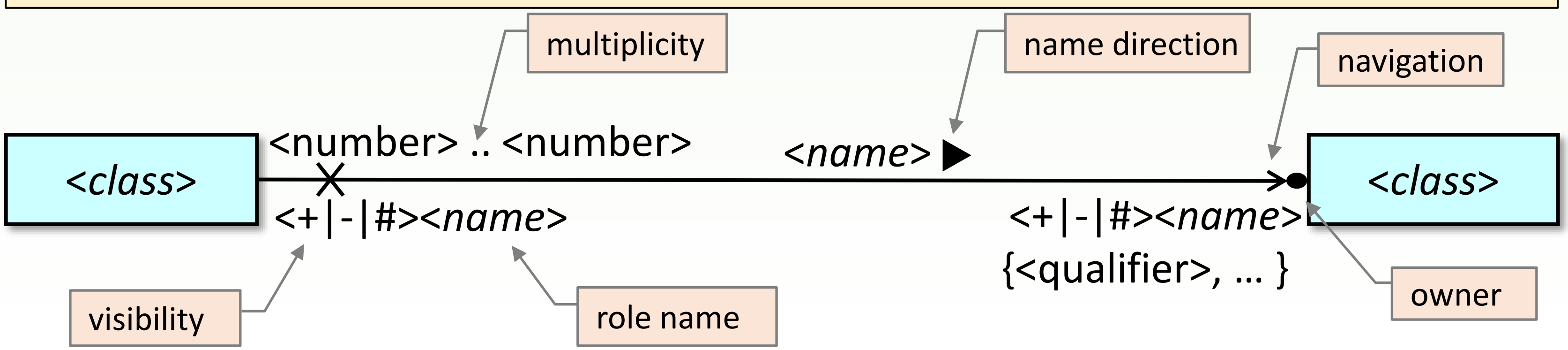


Possible instantiation (population) of the above class diagrams:



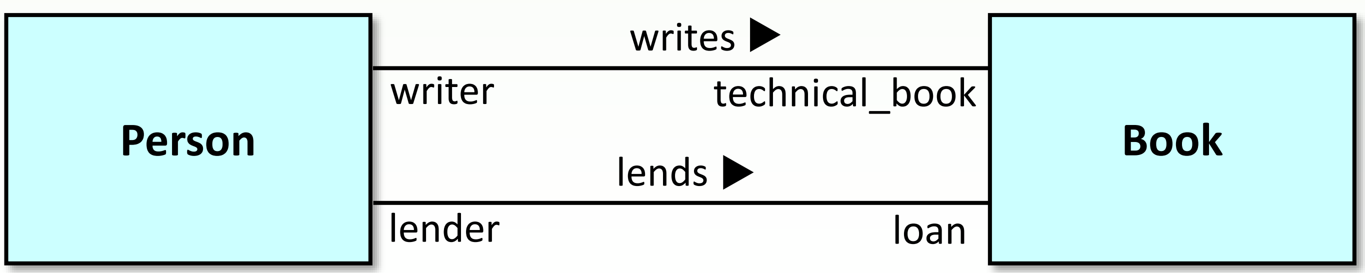
Properties of association

- An association might have several properties, like
  - name
  - name direction
  - multiplicity
  - arity (binary or n-ary associations)
  - navigation
  - end names of the association (role names)
  - visibility and owner of the end names of the association
- If a property is missing, it means that it is not clear yet.



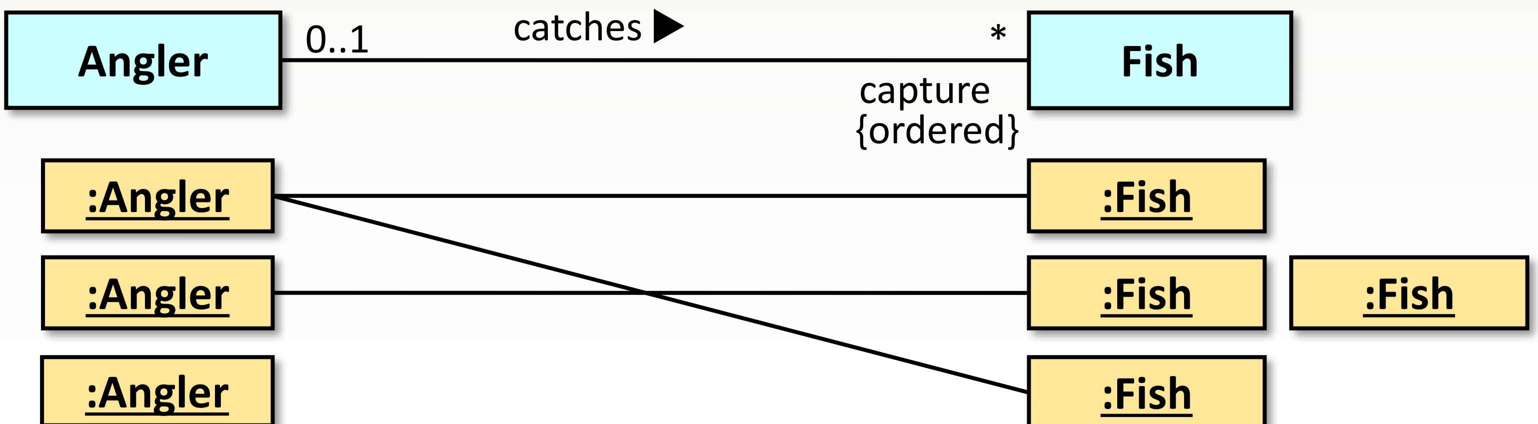
Name and name direction

- Association is usually described by a complex sentence, where the predicate is the name of the association, the rest stand for the end names (role names).
- Binary (between two objects) associations are usually described by
  - a sentence of subjective, predicate, and object, where the end names are the subjective and the object.
  - The black triangle (like an arrow) refers to the object of the sentence: it is the name direction.



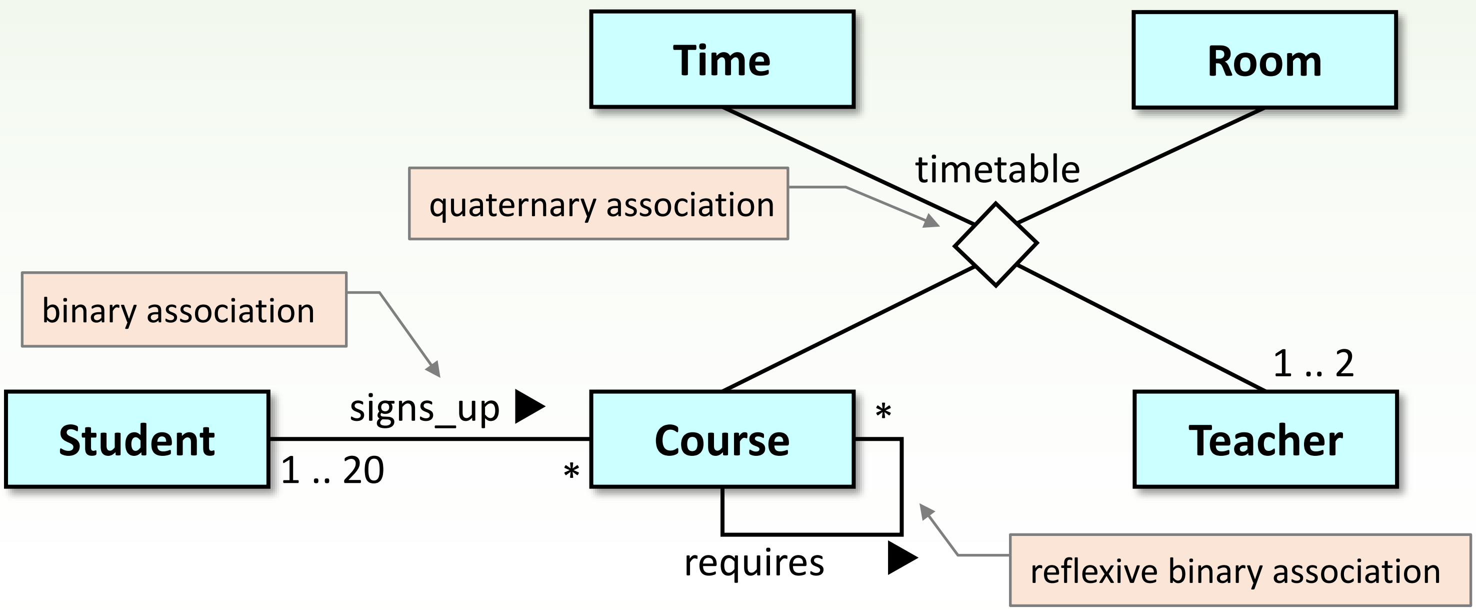
Multiplicity

- Multiplicity shows that how many (min .. max) objects of the class with the multiplicity can establish relationship simultanously with an object of the other class in the association.
  - multiplicity 1 can be skipped
  - instead of 0 .. *, notation * is used, where * is an arbitrary natural number
- If the multiplicity is „many”, it might be prescribed that the objects at the many side are
  - all unique {unique},
  - in a given order {ordered}.



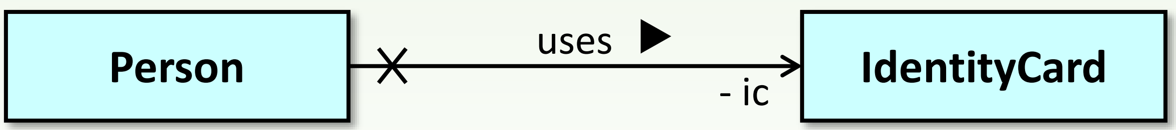
Arity

- Arity means that how many type of objects are connected.
- Until now, only binary associations were shown, where 2 type of objects were connected.
    - Same object may be present in more relationships.
    - Reflexive association connects two objects of the same class.



Navigation

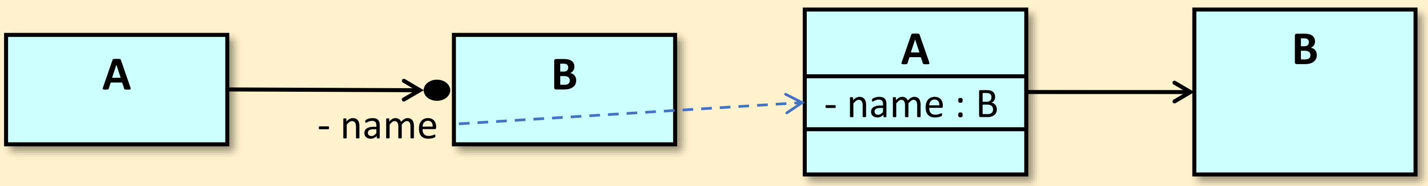
- Navigation shows which object should reach the other effectively.
  - Effective navigation in a given direction is denoted by an arrow at the given end of association.
  - x denotes the non-supported direction of the navigation.
  - Unsigned association refers to an undefined navigation.
- Navigation direction and name direction are different expressions, their directions might differ.




```
class Person {
private:
    IdentityCard *_ic;
public:
    enum Error {ESCAPE};
    IdentityCard showIdentityCard() const {
        if (_id != nullptr) return *_ic;
        else throw ESCAPE;
    }
};
```

Owner of the end name

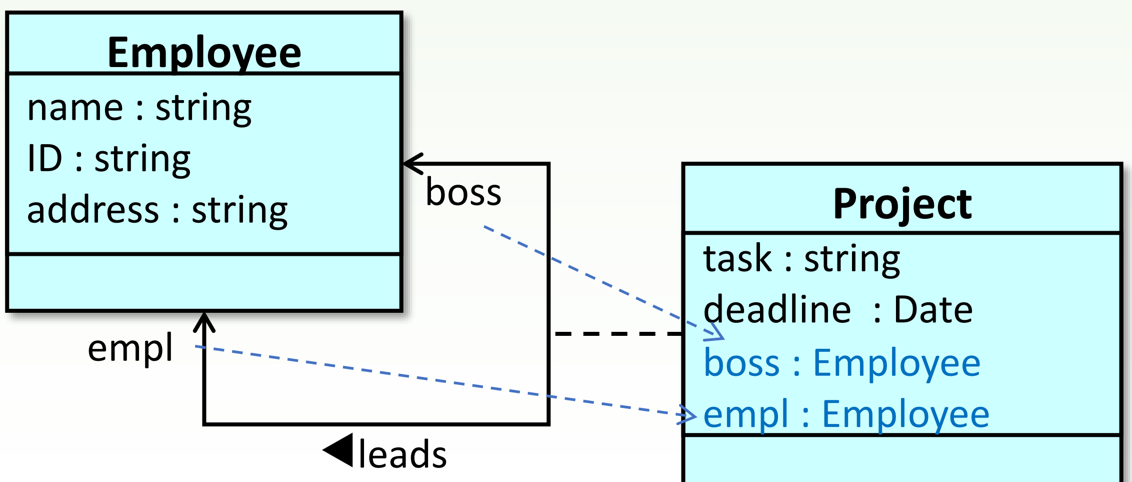
- Objects in a relationship might be referred by the end names (role names). Where are the references stored?
  Who is the owner of the role name?
  - It might be the association itself: the relationship stores the objects and the storage can be accessed by all of the related objects.
  - It might be the other class(es) on the other side: in this case, the attribute name in the other class is the role name. It is denoted by a *black fleck* on the side of the role name.



- Visibility of the role name (private, protected, public) shows if the name is public or, only can be seen by the owner.
- Multiplicity of a class shows if it is a collection or not.

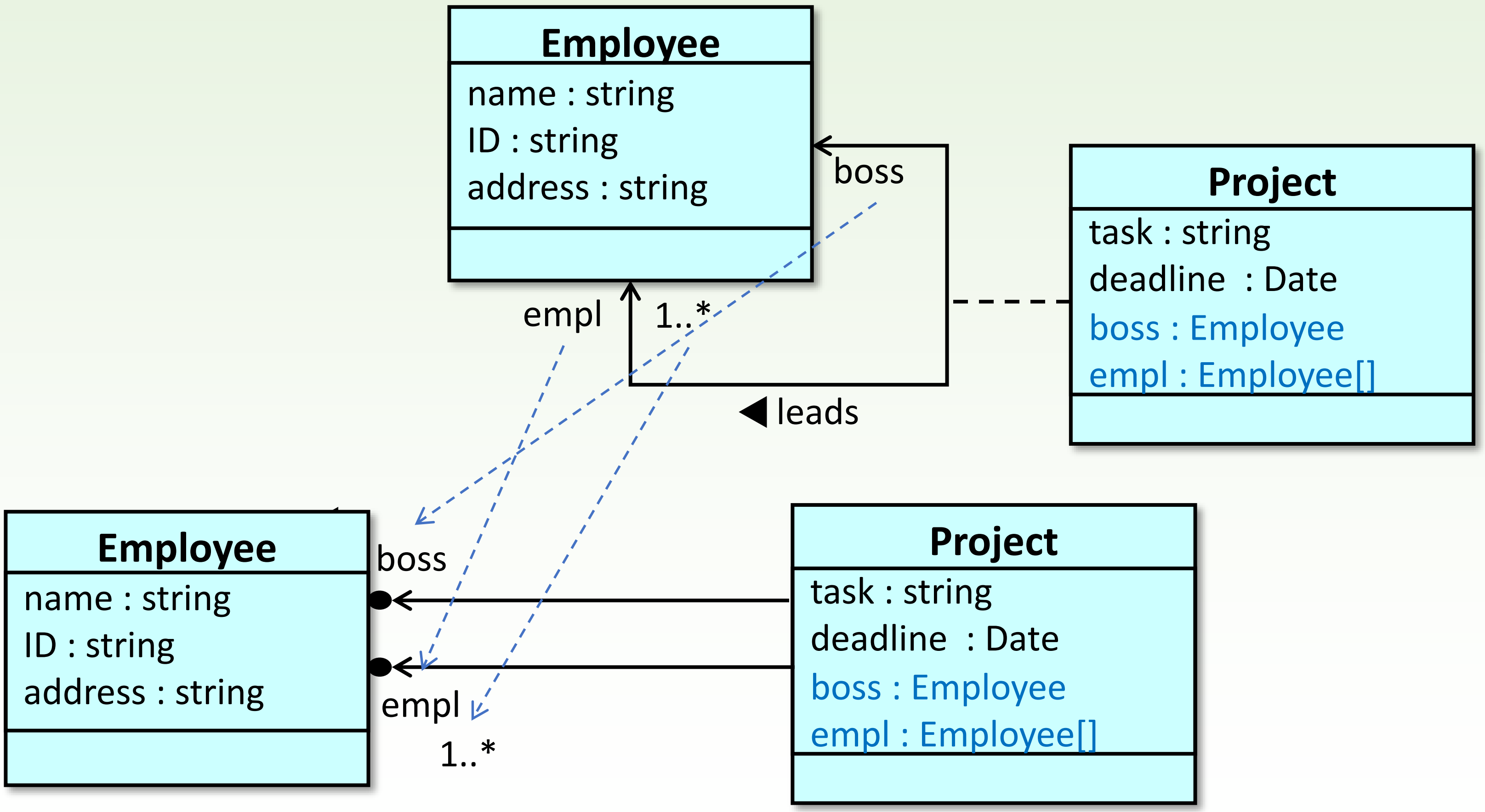
Association class

- In UML, it is possible to define a class to describe a relationship. Instances are relationships which are accessible by the connected objects, thus objects reach the information in it.
- When a role name is owned by the association, the role name is an attribute of the association class.
- Leading OO languages do not support it.



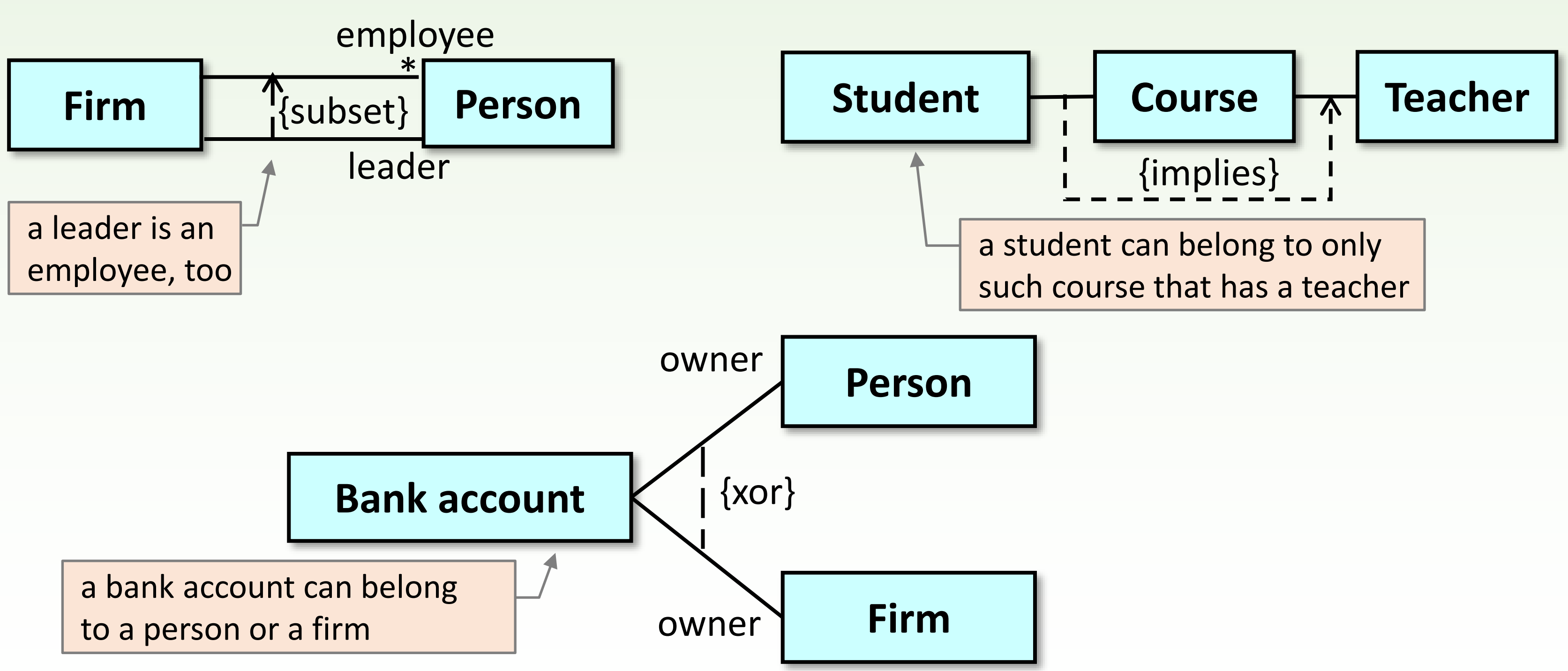
Elimination of an association class

- Replace the association class with a standard class.

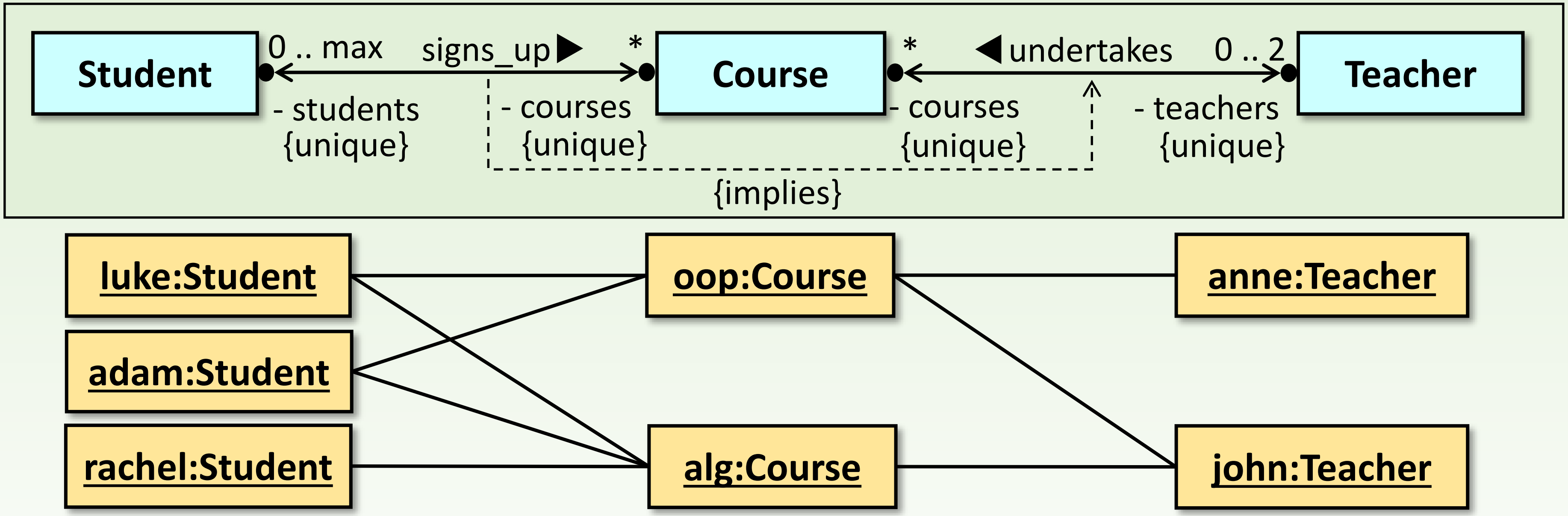


Relationship of associations

- Logical conditions (subset, and, or, xor, implies, …) might be given, which restrict the associations of an object.

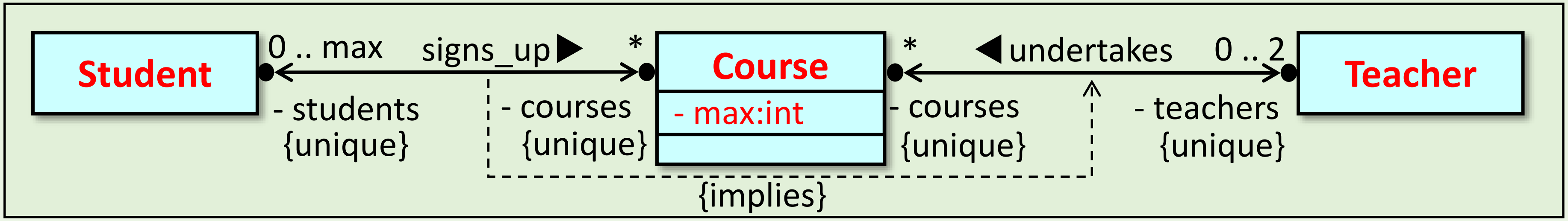


Example




```
Student luke, adam, rachel;
Teacher anne, john;
Course oop(20), alg(22);

anne.undertakes(&oop);
john.undertakes(&oop); john.undertakes(&alg);

luke.signs_up(&alg); luke.signs_up(&oop);
adam.signs_up(&alg); adam.signs_up(&oop);
rachel.signs_up(&alg);
```

Classes

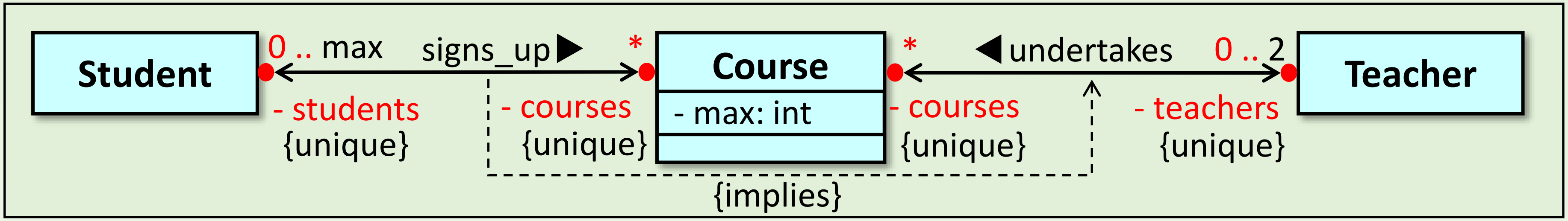



```
class Course {
private:
      unsigned int _max;
      …

public:
      Course(unsigned int max) : _max(max) {}
};
```

```
class Student {
private:
      …
public:
      …
};
```

```
class Teacher {
private:
      …
public:
      …
};
```

Classes

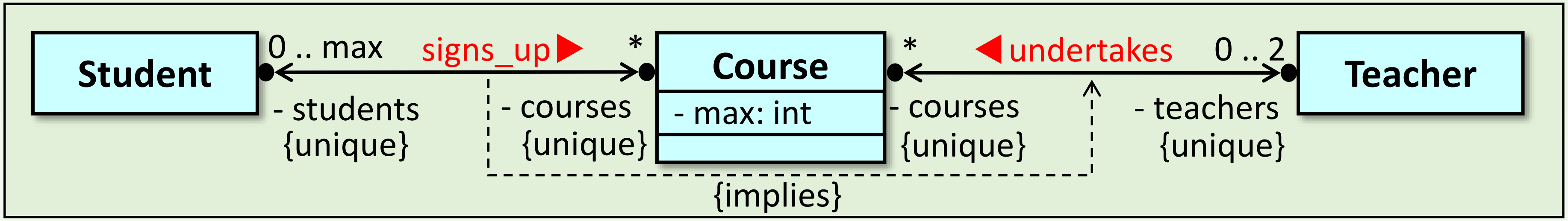



```
class Course {
private:
    unsigned int _max;
    std::vector<Teacher*> _teachers; // 0 .. 2
    std::vector<Student*> _students; // 0 .. max
public:
    Course(unsigned int max) : _max(max) {}
};
```

```
class Student {
private:
    std::vector<Course*> _courses;
public:
    …
};
```

```
class Teacher {
private:
    std::vector<Course*> _courses;
public:
    …
};
```

Classes

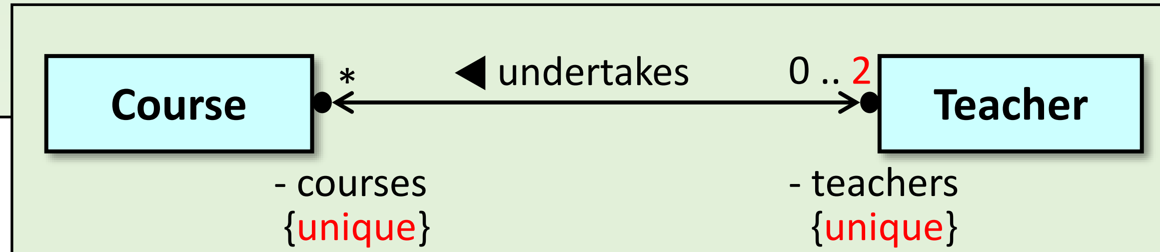



```
class Course {
private:
    unsigned int _max;
    std::vector<Teacher*> _teachers; // 0 .. 2
    std::vector<Student*> _students; // 0 .. max
public:
    Course(unsigned int max) : _max(max) {}
};
```

```
class Student {
private:
    std::vector<Course*> _courses;
public:
    void signs_up(Course* pc);
};
```

```
class Teacher {
private:
    std::vector<Course*> _courses;
public:
    void undertakes(Course* pc);
};
```

Methods



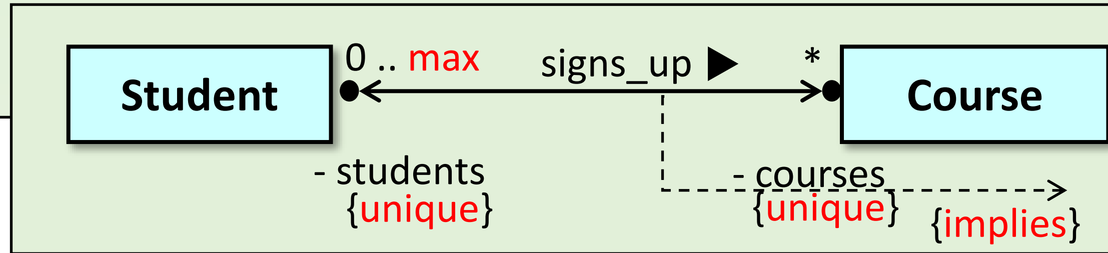

```cpp
class Course {
private:
    unsigned int _max;
    std::vector<Teacher*> _teachers; // 0 .. 2
    std::vector<Student*> _students; // 0 .. max
public:
    Course(unsigned int max) : _max(max) {}
    bool can_lead(Teacher *pt) {
        if (_teachers.size()>=2)
            return false;
        for (Teacher *p : _teachers) {
            if (p==pt) return false;
        }
        _teachers.push_back(pt);
        return true;
    }
};
```

checks the upper bound of the multiplicity

this linear search checks both of the unique conditions

```cpp
class Teacher {
private:
    std::vector<Course*> _courses;
public:
    void undertakes(Course *pc){
        if( pc==nullptr ) return;
        if( pc->can_lead(this) ) {
            _courses.push_back(pc);
        }
    }
};
```

Methods




```
class Course {
private:
    unsigned int _max;
    std::vector<Teacher*> _teachers; // 0 .. 2
    std::vector<Student*> _students; // 0 .. max
public:
    Course(unsigned int max) : _max(max) {}
    bool can_lead(Teacher *pt) { … }
    bool has_teacher() const { return _teachers.size()>0; }
    bool can_sign_up(Student *ps) {
        if (_students.size()>=_max )
            return false;
        for (Student *p : _students) {
            if (p==ps) return false;
        }
        _students.push_back(ps);
        return true;
    }
};
```

checks the implies condition

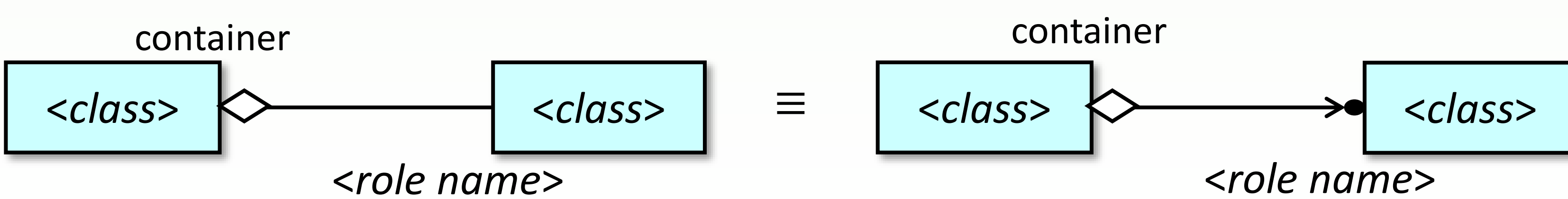
this linear search checks both of the unique conditions

checks the upper bound of the multiplicity

```
class Student {
private:
    std::vector<Course*> _courses;
public:
    void signs_up(Course *pc){
        if( pc==nullptr
            || !pc->has_teacher() )
                return;
        if( pc->can_sign_up(this) )
            _courses.push_back(pc);
    }
};
```

Aggregation

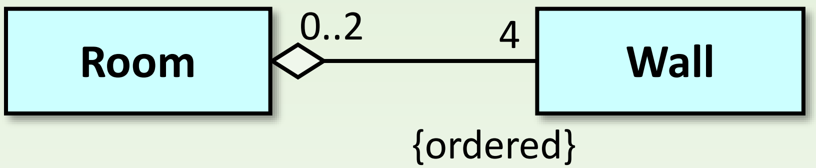
- It is a binary association expressing that an object contains or owns another object:
  - It is asymmetric, non-reflexive, transitive, and cannot contain loop neither indirectly (an object cannot contain itself).
  - An object can be part of many other objects – even simultaneously, too –, and if the container is destroyed, the contained object can live on.
- We agree that for lack of notation,
  - the relationship is navigated in the direction of the contained class,
  - role name of the contained class belongs to the container.



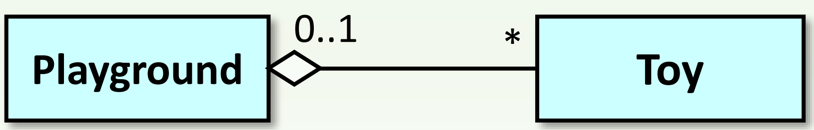
Composition

- Composition is a special aggregation where the contained objects can belong to only one container, and the contained objects cannot exist by themselves.
- Several explanations are known, e.g.: the contained object
  - is always part of a container,
  - cannot change its owner, and
  - can only be created and destroyed by the container: usually its constructor instantiates it and its destructor destroys it.

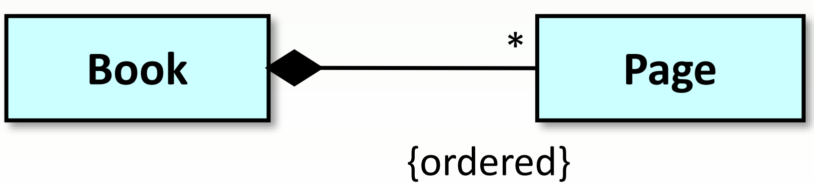
Examples



A wall may belong to two rooms
(a wall may survive a room).

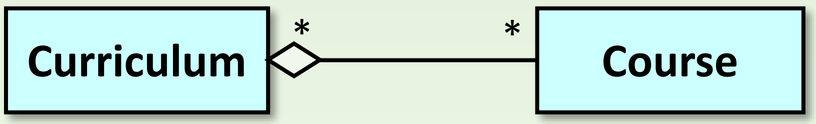


The toys belong to one playground, but they can be transferred to another playground. If a playground is closed, the toys are still usable.

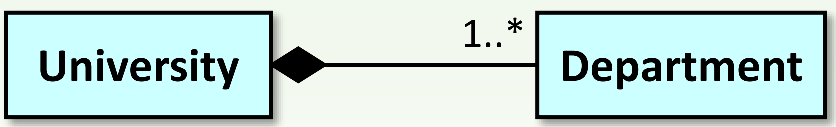


The pages cannot get into another book. A page may fall out of the book, but usually, the page becomes unusable.

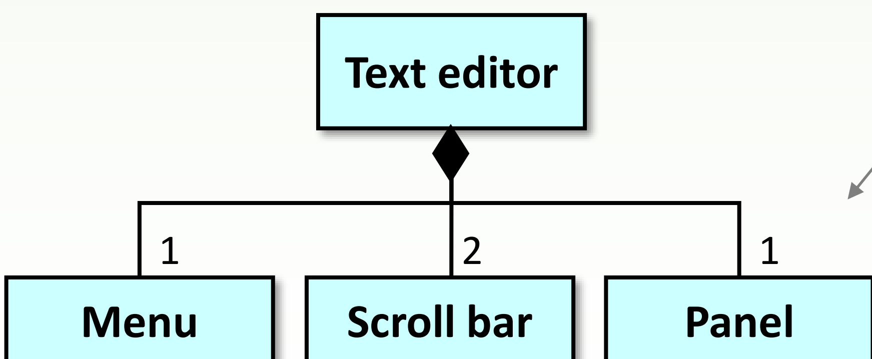
Examples



A course may belong to more curricula (a course may survive a curriculum).



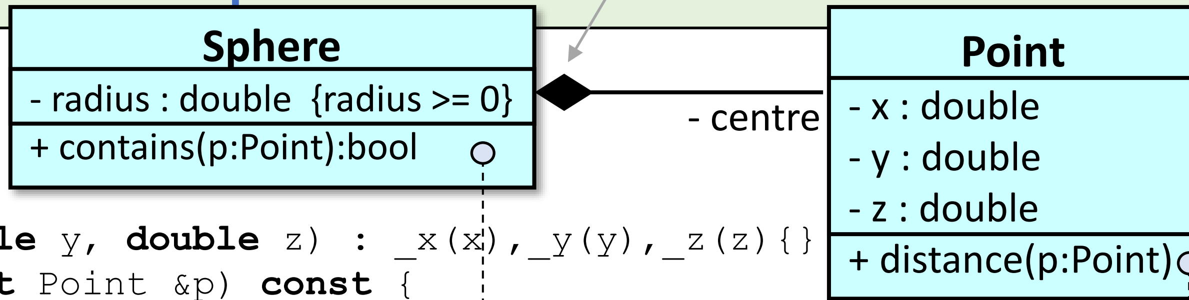
The departments belong to one university. If the university is closed, the departments are closed, too.



Accessories of an editor are created when the editor is created.
The accessories are destroyed when the editor is destroyed.
These accessories belong to one editor.

Ex.: point in a sphere

The centre belongs to the sphere, it is created and destroyed together with the sphere.




```
class Point {
private:
    double _x, _y, _z;
public:
    Point(double x, double y, double z) : _x(x),_y(y),_z(z){}
    double distance(const Point &p) const {
        return sqrt(pow(_x-p._x,2) + pow(_y-p._y,2) + pow(_z-p._z,2));
    }
};
```

return centre.distance(p) ≦ radius

return sqrt((x–p.x)^2+(y–p.y)^2+(z–p.z)^2)

```
class Sphere{
private:
    Point  _centre;
    double _radius;
public:
    enum Errors{ILLEGAL_RADIUS};
    Sphere(const Point &c, double r): _centre(c), _radius(r) {
        if (_radius<0.0) throw ILLEGAL_RADIUS;
    }
    double contains(const Point &p) const {
        return _centre.distance(p) <= _radius;
    }
};
```

composition

Copies Point *c* for the automatically created centre: in this case, the centre and the point in c are two different objects.

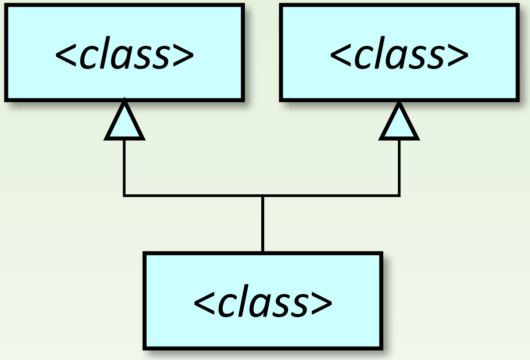
```
Point  p(-12.0, 0.0, 23.0);
Point  c(-12.3, 0.0, 23.4);
Sphere s1(c, 1.0);
cout << s1.contains(p) << endl;
cout << c.distance(p) << endl;
```

How to implement a relationship

- In case of association, one of the objects creates the relationship by one of its methods (named after the association's name or by a constructor). The method gets the other objects' reference as parameters or instantiates them.
- In case of aggregation, the container creates the relationship.
- In case of composition, the container's constructor creates the relationship.
  - If the contained element can be replaced, it is done by a method.
  - In stricter explanations, the constructor instantiates and the destructor destroyes the contained element.

Inheritance

- If an object is similar to another, (it has the same attributes and methods), then its class may be inherited from the classes of the similar objects: inherits their properties, and it might modify and augment them.



- In modeling, there are two reasons for inheritance:
  - Generalization: to create a base class (superclass) to describe common properties of similar, already existing classes.
  - Specialization: to create a subclass by inheritance.

5

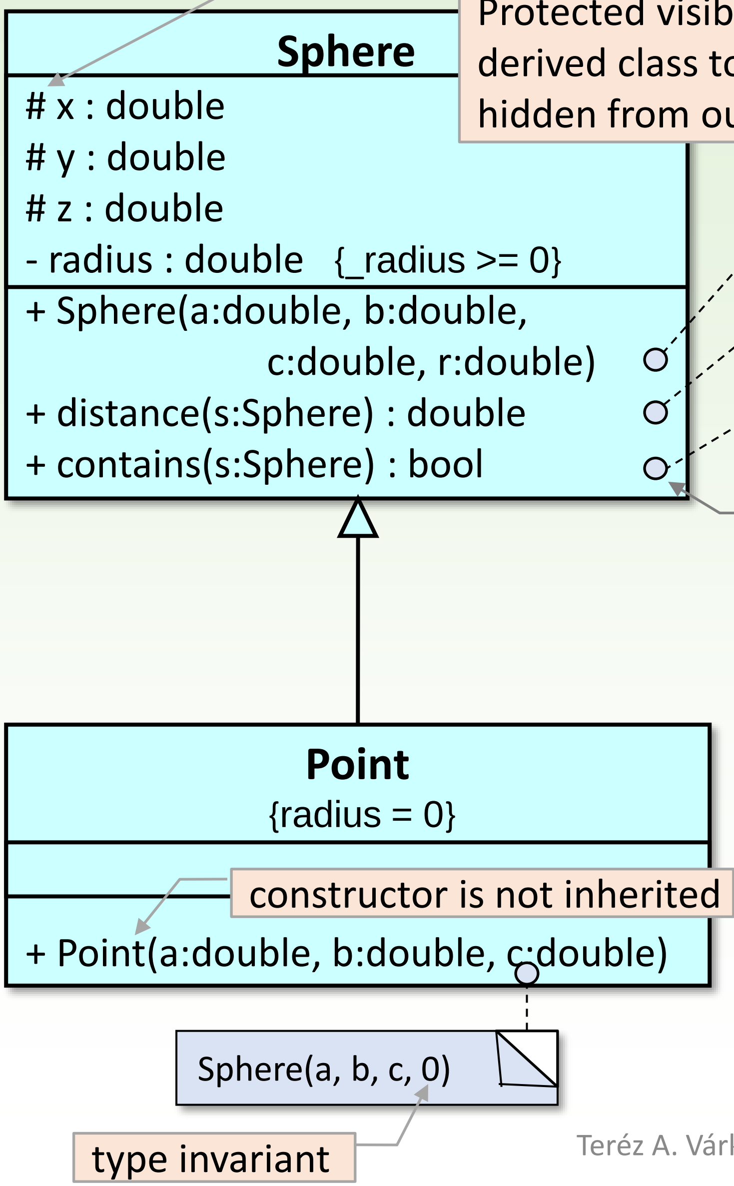
A well-known criterion of object-orientation is inheritance: classes may be derived from already existing classes.
Object of a subclass might be assigned to a variable of the base class.

Inheritance and visibility

- In the child class, public and protected attributes of the parent might be referred. Private attributes cannot be accessed, but indirectly, through inherited methods they might be modified.
- Inheritance may be public, protected, or private.
  - Public, if the visibilities of the protected and public attributes are inherited as they are defined in the base class. (In UML this is default, in language C++, not.)
  - Protected, if the public and protected members become protected in the child.
  - Private, if the public and protected members become private in the child.

Ex.: point from sphere

Single responsibility
O
Liskov substitution
I
D



Protected visibility makes it possible for the derived class to use them directly. They are hidden from outside.

x, y, z, radius := a, b, c, r

return sqrt((x–s.x)^2+(y–s.y)^2+(z–s.z)^2) – this.radius – s.radius

return distance(s) + 2 · s.radius ≦ 0

Object of a derived class might be assigned to a variable of the base class.

Because of the inheritance, functions *distance*() and *contains*(), might be called for points, too, and their parameters may be points, too (not just spheres):

```
Sphere s1, s2; Point p1, p2;
… s1.contains(s2) …
… s1.contains(p2) …
… p1.contains(s2) …
… p1.contains(p2) …
```

All of these have to work correctly.

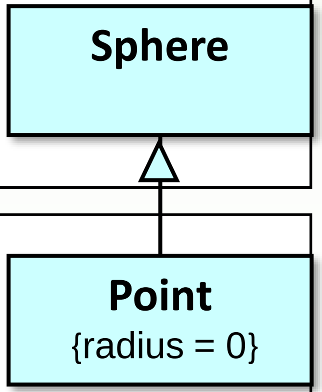
C++ : point from sphere

```
Point  p(0,0,0);
Sphere s(1,1,1,1);

cout << s.contains(p) << endl;
cout << s.distance(p) << endl;
cout << s.contains(s) << endl;
cout << s.distance(s) << endl;
cout << p.contains(s) << endl;
cout << p.distance(s) << endl;
cout << p.contains(p) << endl;
cout << p.distance(p) << endl;
```

```
class Sphere{
protected:
    double _x, _y, _z;
private:
    double _radius;
public:
    enum Errors{ILLEGAL_RADIUS};
    Sphere(double a, double b, double c, double r):
        _x(a),_y(b),_z(c),_radius(r) { if(_radius<0.0) throw ILLEGAL_RADIUS; }
    double distance(const Sphere& s) const
        { return sqrt(pow((_x-s._x),2) + pow((_y-s._y),2) + pow((_z-s._z),2))
                 -_radius - s._radius; }
    bool contains(const Sphere& s)  const
        { return distance(s) + 2 * s._radius <= 0; }
};
```

public inheritance

```
class Point : public Sphere {
public:
    Point(double a, double b, double c) : Sphere(a, b, c, 0.0) {}
};
```



Example: sphere from poin

Single responsibility
O
Liskov substitution
Interface segregation
Depedency inversion

These calls cannot be expounded obviously, as it is decided runtime if variables *p* and *this* refer to a point or a sphere.
this.radius() and *p.radius*() depend on the type of them.

Point

\# x : double
\# y : double
\# z : double

\+ Point(double, double, double)
\+ distance(p:Point) : double
\+ radius() : double { virtual }

return sqrt((x−p.x)^2+(y−p.y)^2+(z−p.z)^2) − radius() − p.radius()

return ~~sqrt((x−p.x)^2+(y−p.y)^2+(z−p.z)^2)~~

return 0.0

If it is called on spheres, it does not give correct result, it violates the Liskov-principle.

In OO languages, if radius() is virtual, then when it is called on a pointer or a reference variable (e.g. variable *this*), that version is run where the pointer or reference points at.

Sphere

\# radius : double {radius >= 0}

\+ Sphere(a:double, b:double, c:double, r:double)
\+ contains(p:Point) : bool
\+ radius() : double { override }

Point(a,b,c); radius := r

return distance(p) + 2·p.radius() ≦ 0

return ~~distance(p) ≦ radius~~

return radius

C++ : sphere from point

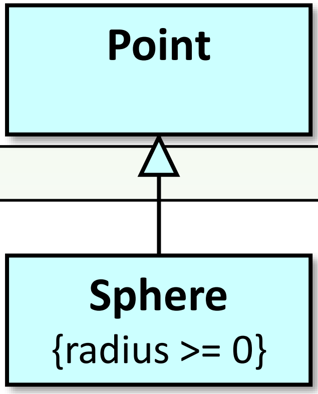
```
Point  p(0,0,0);
Sphere g(1,1,1,1);

cout << p.distance(p) << endl;
cout << g.distance(p) << endl;
cout << p.distance(g) << endl;
cout << g.distance(g) << endl;

cout << g.contains(p) << endl;
cout << g.contains(g) << endl;
```

this->radius()

reference variable

```
class Point {
protected:
    double _x, _y ,_z;
public:
    Point(double a, double b, double c) : _x(a), _y(b), _z(c) {}
    double distance(const Point &p) const
        { return sqrt(pow((_x-p._x),2)+pow((_y-p._y),2)+pow((_z-p._z),2))
                - radius() - p.radius();}
    virtual double radius() const { return 0.0; }
};
```

virtual method




```
class Sphere : public Point {
private:
    double _radius;
public:
    enum Errors{ILLEGAL_RADIUS};
    Sphere(double a, double b, double c, double r) : Point(a,b,c), _radius(r)
        { if(_radius<0.0) throw ILLEGAL_RADIUS; }
    bool contains(const Point &p) const { return distance(p)+2*p.radius()<=0; }
    double radius() const override { return _radius; }
};
```

overridden method

Polymorphism and dynamic binding

- If a method of the parent is overridden in a child (override), that method has several ”shapes” (it is polymorphic).
- An instance of a child might be assigned to a variable of the parent, so it has to be decided runtime if the variable refers to an instance of the parent or the child (late or dynamic binding).
- If a polymorphic virtual method of the parent is called on a pointer or reference variable, then the type of the object (child or parent) determines which version of that method is run (runtime polymorphism).

6 Typical criterion of object-orientation is runtime (subtype) polymorphism together with dynamic binding.