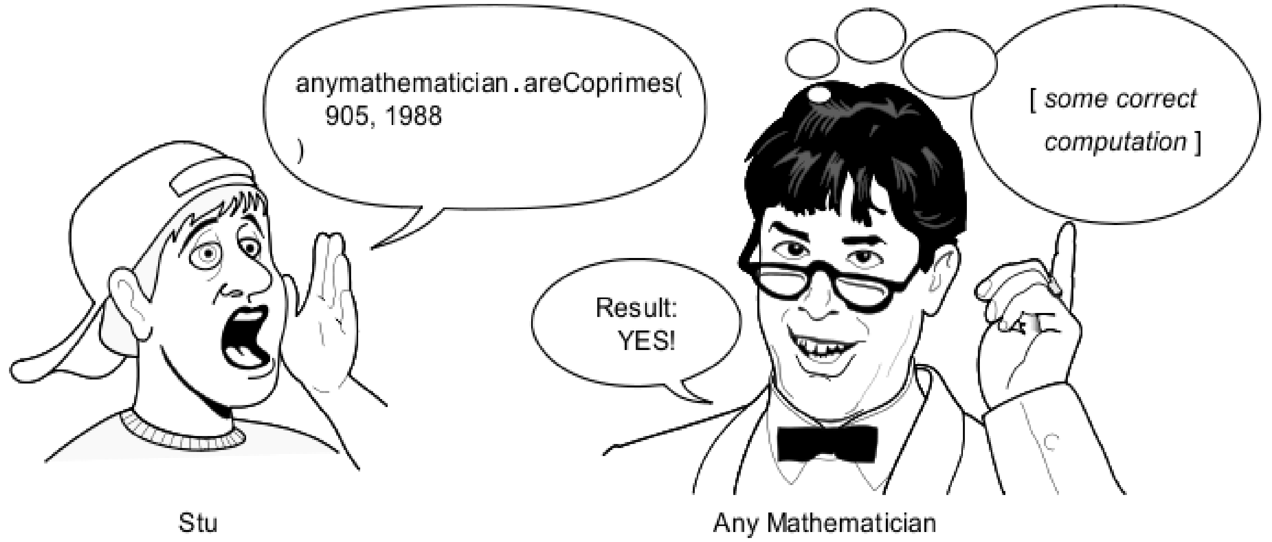
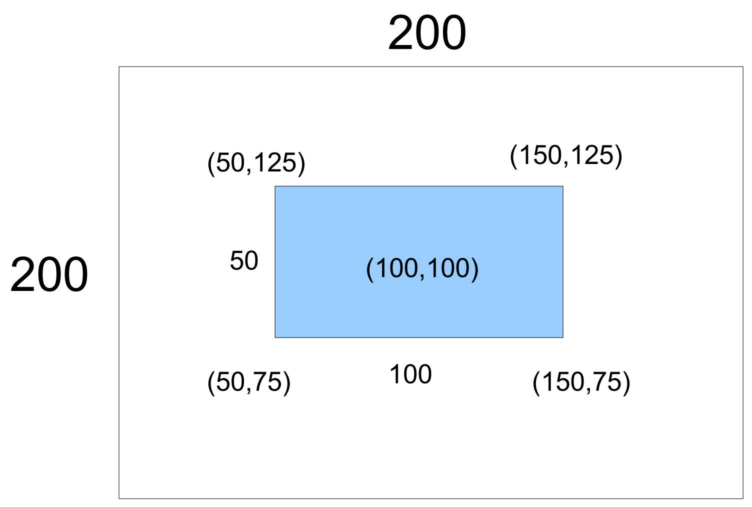
Object Associations

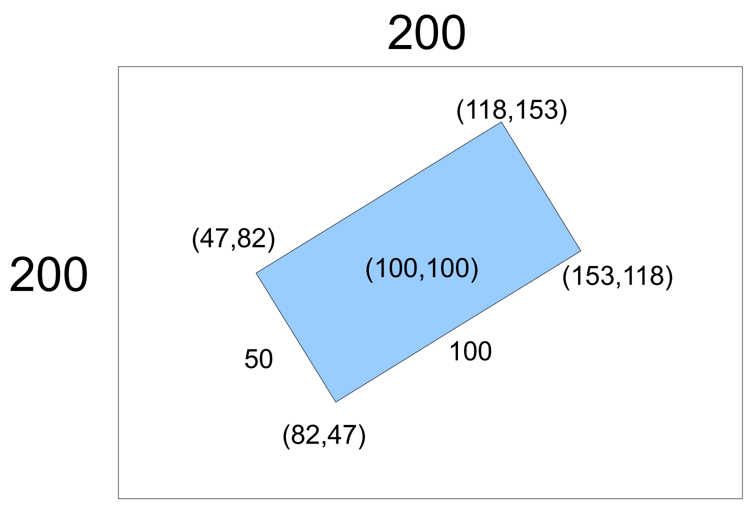
Collaboration among objects

- Objects collaborate in order to solve a meaningful problem
- This collaboration leads to dependencies, associations and other relationships
- The fundamental way of reusing behavior



Source: *Software Engineering, Ivan Marsic*





Rotation Matrix

$$\begin{bmatrix}\cos\theta & -\sin\theta \\ \sin\theta & \cos\theta\end{bmatrix} \quad \begin{bmatrix}x \\ y\end{bmatrix}$$

Rotating Top Left point (50, 125) by an angle 45 degree

1. Center point on (0,0)

$$\begin{bmatrix}50 \\ 125\end{bmatrix} - \begin{bmatrix}100 \\ 100\end{bmatrix} = \begin{bmatrix}-50 \\ 25\end{bmatrix}$$

2. Apply rotation matrix

$$\begin{bmatrix}\cos 45 & -\sin 45 \\ \sin 45 & \cos 45\end{bmatrix} \begin{bmatrix}-50 \\ 25\end{bmatrix} = \begin{bmatrix}-53 \\ -18\end{bmatrix}$$

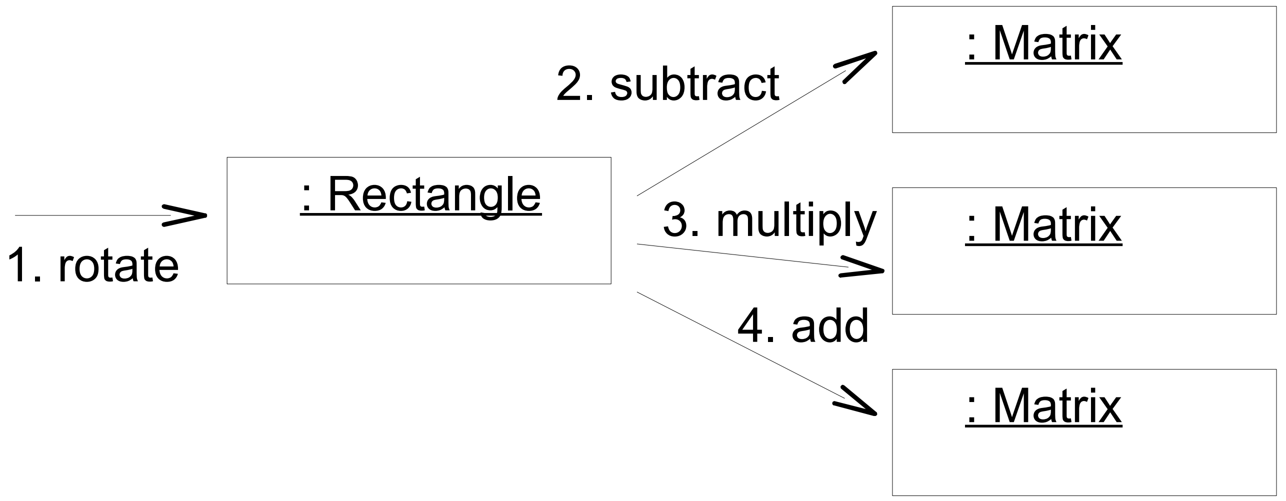
3. Re-adjust according to center

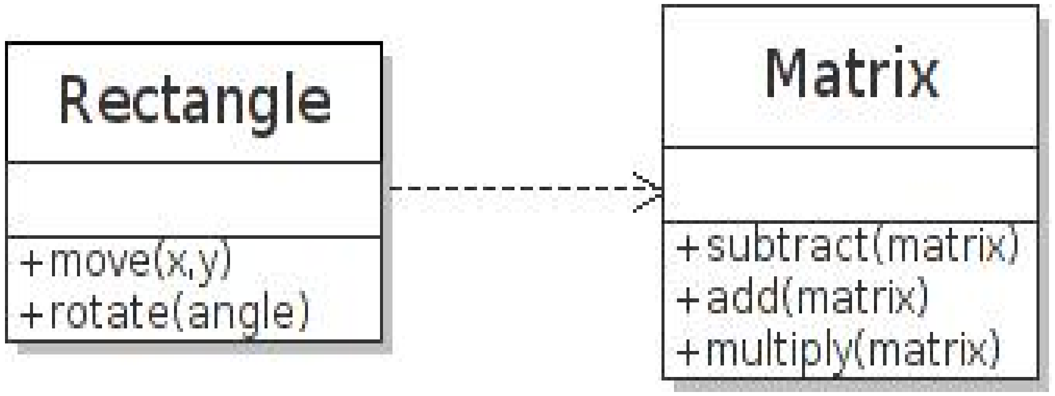
$$\begin{bmatrix}-53 \\ -18\end{bmatrix} + \begin{bmatrix}100 \\ 100\end{bmatrix} = \begin{bmatrix}47 \\ 82\end{bmatrix}$$

```
class Rectangle {

    private:
        int x;
        int y;
        int width;
        int height;
        int angle;

        Matrix *  getPoints();
        Matrix *  getCenter();
        void      redraw(...);

    public:
        Rectangle(...)
        void move(int x,int y);
        void rotate(int angle);
        …
};
```

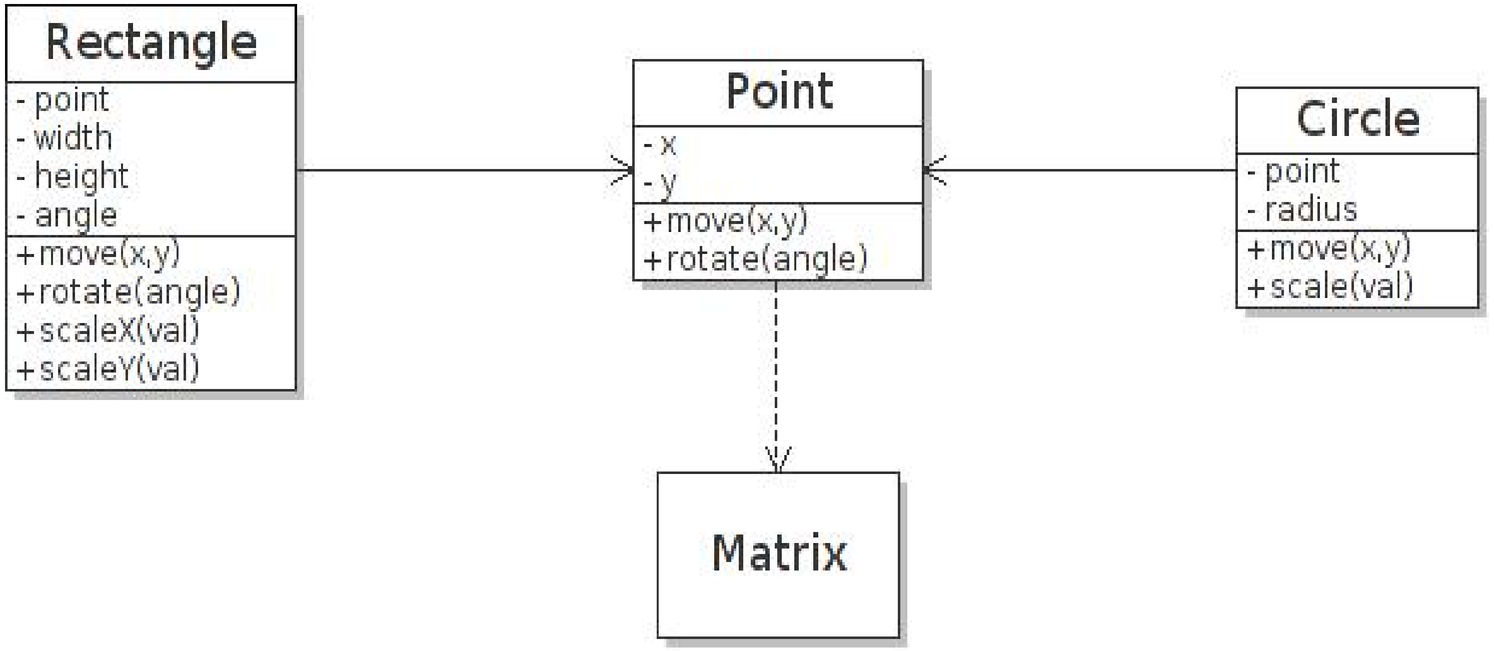
```
Rectangle::rotate(int angle){

  Matrix * rm = new RotationMatrix (angle);

  Matrix **  points = getPoints();
  Matrix * center = getCenter();

  for (int i=0; I < 4; i++){

    // step-1: center to (0,0)
    points[i] = points [i] –> subtract (center);

    // step-2: apply rotation matrix
    points[i] = rm -> multiply (points[i] );

    // step-3: readjust center
    points[i] = points[i] -> add (center);
  }

  redraw(points);
}
```

Collaboration & Dependency





Associations



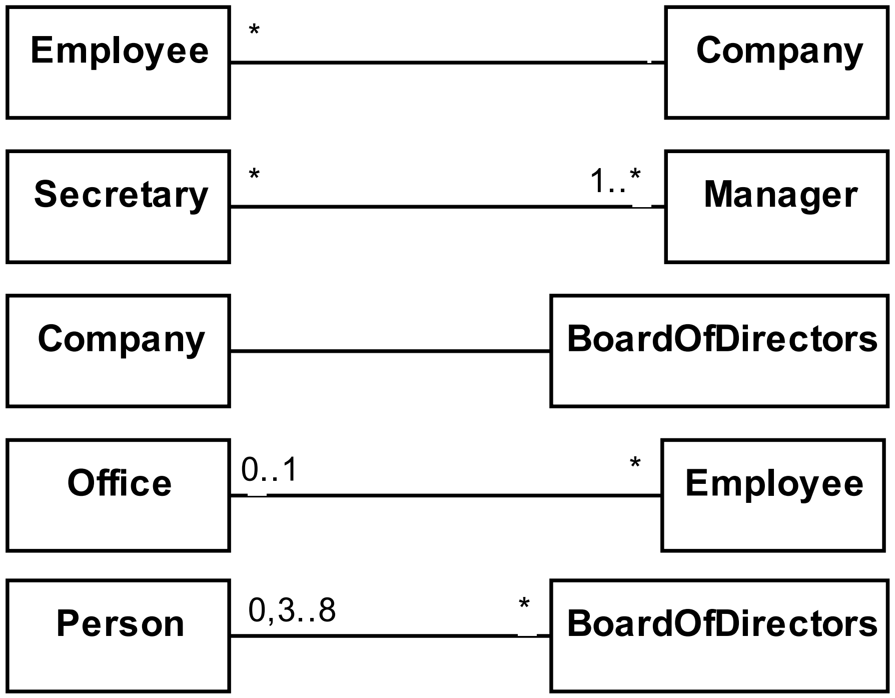
- Associations define structural relationships between objects via their respective classes
  - Shows connection
  - Generally a pointer / reference is maintained for the objects sharing connection

```
class Rectangle {

    private:
        Point * center;
        int width;
        int height;
        int angle;

        Point **  getPoints();
        void      redraw(...);

    public:
        Rectangle(...)
        void move(int x,int y);
        void rotate(int angle);
        …
};
```

```
class Point {

  private:
    int x;
    int y;

  public:
    Point(...)
    void move(int x,int y);
    void rotate(int angle);
            …
};
```

```
class Circle {

  private:
    Point * center;
    int radius;

  public:
    Circle(...)
    void move(int x,int y);
            …
};
```

Associations Examples



Many employees of a company

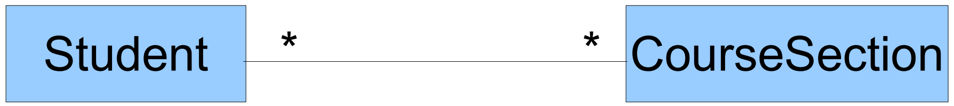
Many secretaries of one or more managers

Company and its board of directors

Many employees of 0 or 1 office

1 person on multiple boards, board having no or otherwise 3 – 8 members

Student registers in course-section

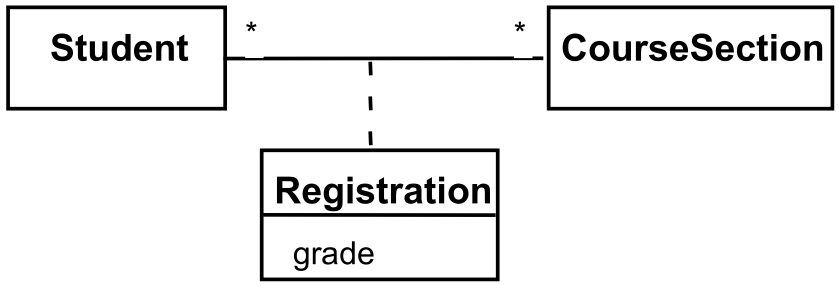


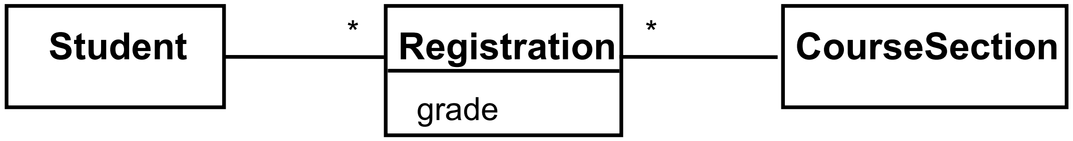
```
class Student {

    private:
        List<Course*> courses;
        …

    public:
        void addCourse(Course* c);
        void dropCourse(Course* c);
        …
};

void Student::addCourse(Course* c){
    if (! courses->exists(c)) {
        courses.add(c);
        c->addStudent(this);
    }
}

…
```

```
class CourseSection {

    private:
        List<Student*> students;
        …

    public:
        void addStudent(Student* s);
        void removeStudent(Student* s);
        …
};

void Course::addStudent(Student* s){
    if (! students->exists(s)) {
        students.add(s);
        s->addCourse(this);
    }
}

…
```

Association Classes

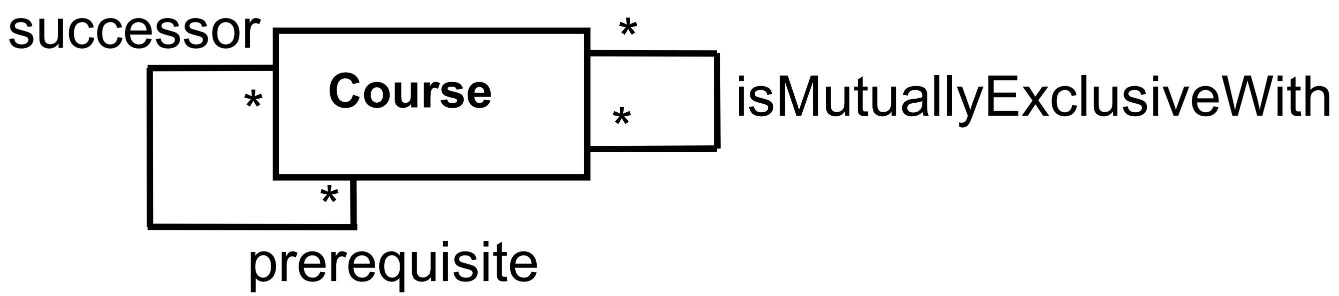
- Sometimes an attribute cannot be placed in either of the associated classes
- Association itself is modeled as class





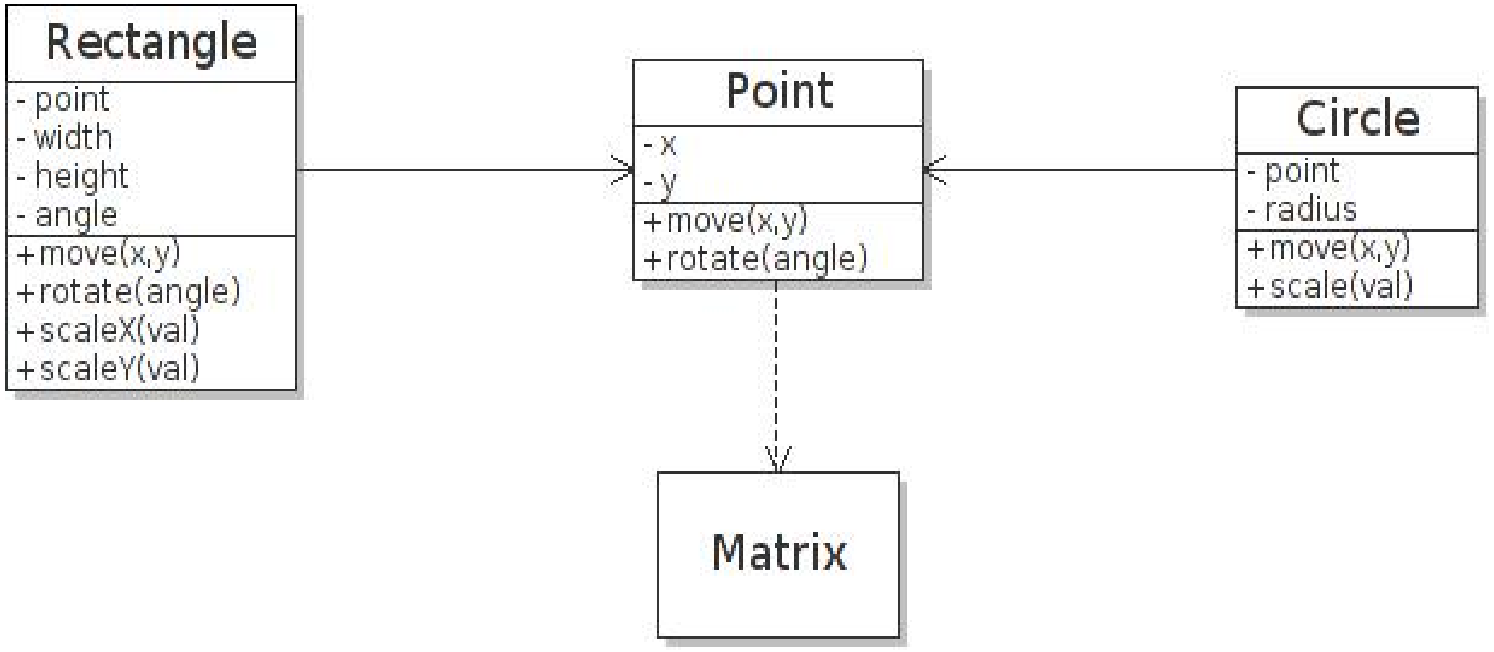
Reflexive Association

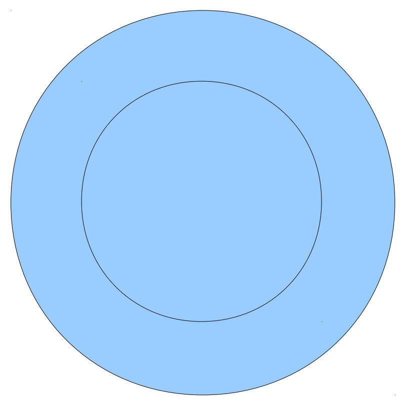
- Objects of a class can be associated to objects of same type



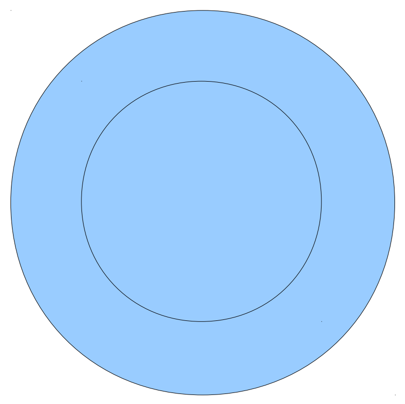
Implementation Issues

- Sharing or mutual exclusion





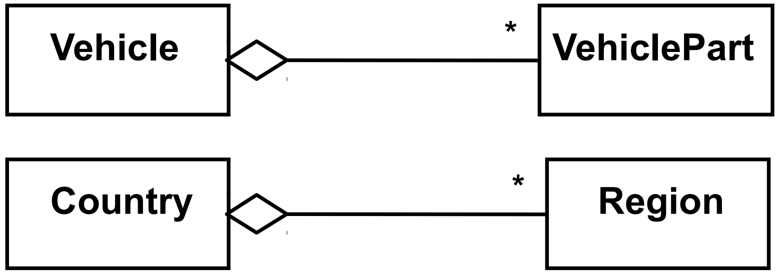
Shared



Exclusive

Aggregation

- Mutual Exclusion leads to part-whole relationship also termed as aggregation
  - Diamond symbol used as notation
  - Points towards whole, not the part



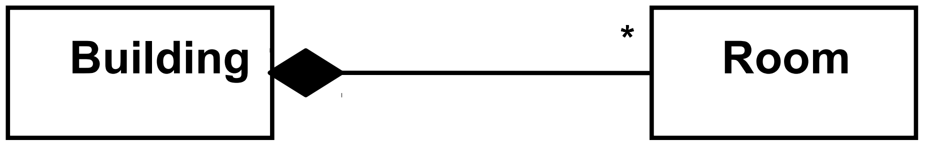
Implementation issues

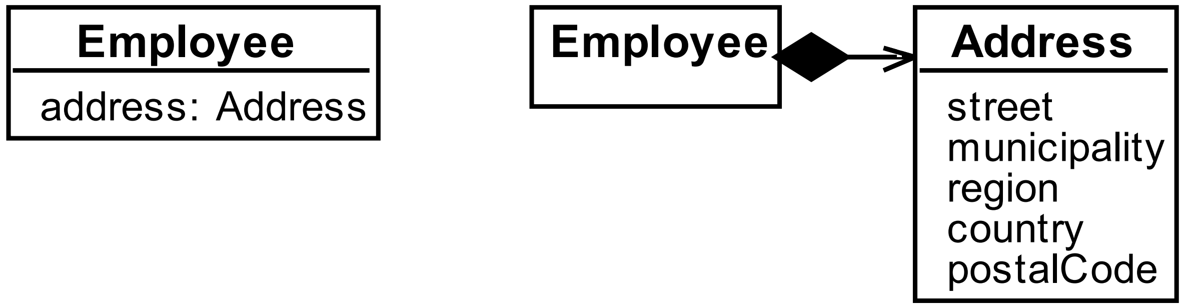
- Object lifeline of part and whole
  - dependent
    - Whole destroys, then part also destroys
  - Independent
    - Whole destroys, but part stays

```
class Circle {

  private:
    Point * center;
    int radius;

  public:
    Circle(...); // instantiate center point here
    ~Circle(); // delete center point
              …
};
```

Composition

- Strong aggregation
  - Whole is destroyed then part destroys also
  - Black diamond is used for notation


Building
*
Room


Employee
address: Address
Employee
Address
street
municipality
region
country
postalCode