CS 213 – Software Methodology
Spring 2017

Sesh Venugopal

Lecture 13 – Mar 2

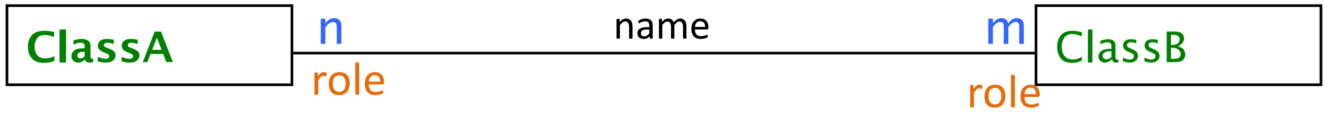
UML Class Diagram - II

Association and Multiplicity

- An association is a general relationship between two classes, with options for name of association, and number of instances (multiplicity) of participation of each class

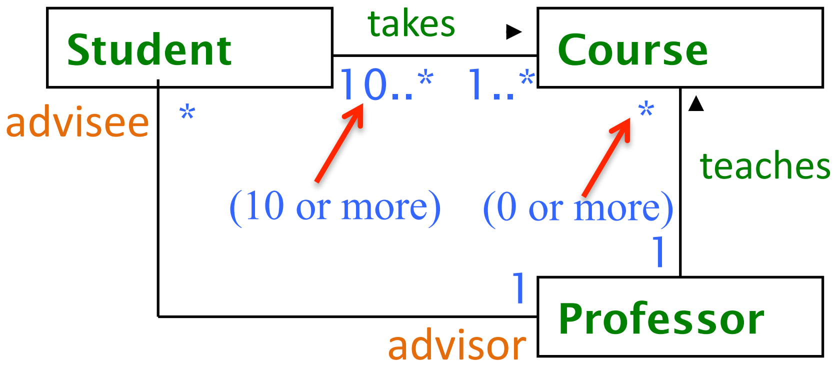
Each instance of ClassB *is associated with n instances of* ClassA

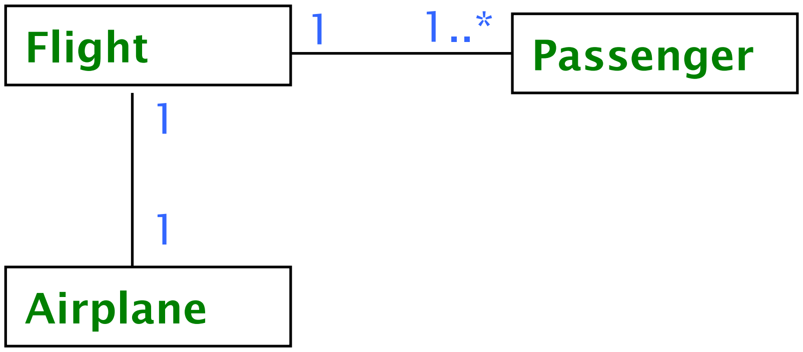
Each instance of ClassA *is associated with m instances of* ClassB



Association and Multiplicity: Examples

Association and Multiplicity: Examples

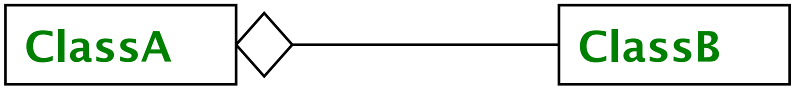




- Multiplicity can also be specified as one of the values an enumerated set such as 1, 3..5

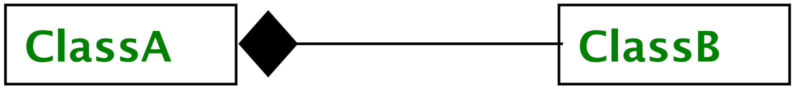
Aggregation and Composition

- Aggregation is a special kind *of association that represents a has-a or* whole-parts relationship – the *whole* is the aggregate class instance, and the *parts* are the component class instances



Each instance of ClassA *aggregates one or more instance of* ClassB

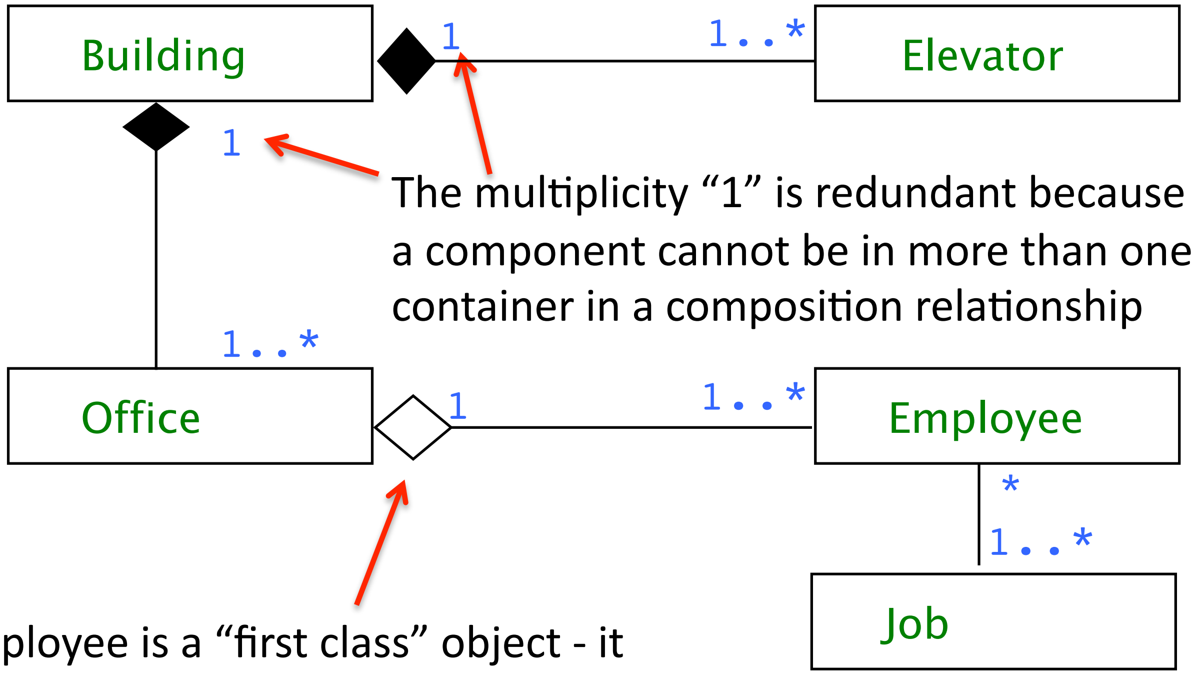
- Composition is a stronger form of aggregation, in which the *components* live or die with the containing class (the whole)—a deletion of the whole will lead to the deletion of the parts (an object may be a part of only one composite at a time)



Each instance of ClassA *is composed of one or more instance of* ClassB

Aggregation and Composition: Example

Aggregation and Composition: Example



The multiplicity "1" is redundant because a component cannot be in more than one container in a composition relationship

Employee is a "first class" object - it has an independent existence. Even if the Office object goes away, Employee objects will still remain. Hence aggregation instead of composition.

Aggregation and Composition

One possible implementation of a composition is to define the composed object (e.g. elevator/office) as an inner class of the composing object (e.g. building)

Elevator and Office are defined as non-static inner classes – creating an object of either requires a Building object

Deleting the whole must result in deleting the parts – implementation wise this applies to languages that do NOT have garbage collection (e.g. C++) because memory for components must be explicitly freed

```
public class Building {
    private class Elevator {
       ...
    }

    private class Office {
       private ArrayList<Employee>
             employees;
       ...
    }

    private Elevator[] elevators;
    private Office[] offices;

    public Building(int enum,
                    int onum) {
         elevators = new Elevator[enum];
         offices = new Office[onum];
         ...
    }
    ...
}
```

Aggregation and Composition

Employee contains a reference to the Office instance with which an employee is associated. (This is not shown in the UML, so it is optional – later on we will see how to make this explicit in the UML)

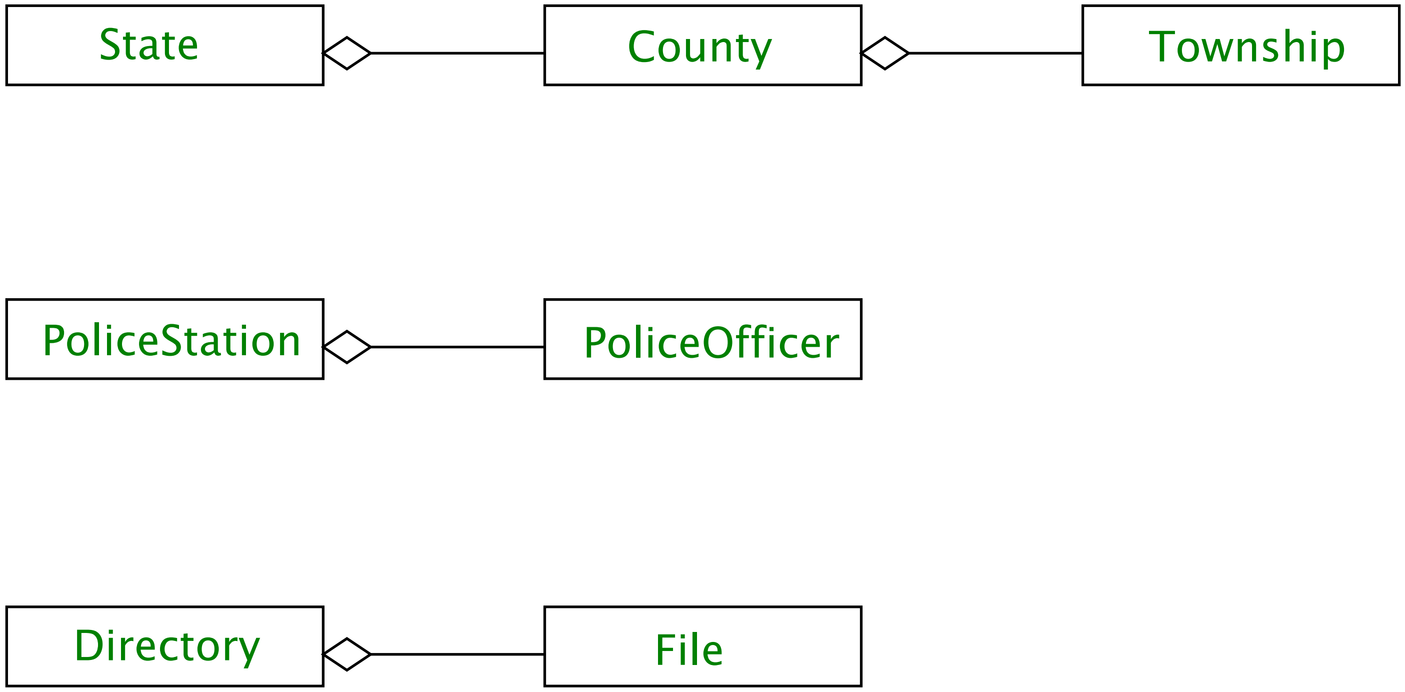
```
public class Employee {
   private String name;
   ...
   private Office office;
}
```

Similar implementation in the Job class, which holds a reference to all employees (could be none) who hold this job

```
public class Job {
   private String title;
   ...
   private ArrayList<Employee>
       employees;
}
```

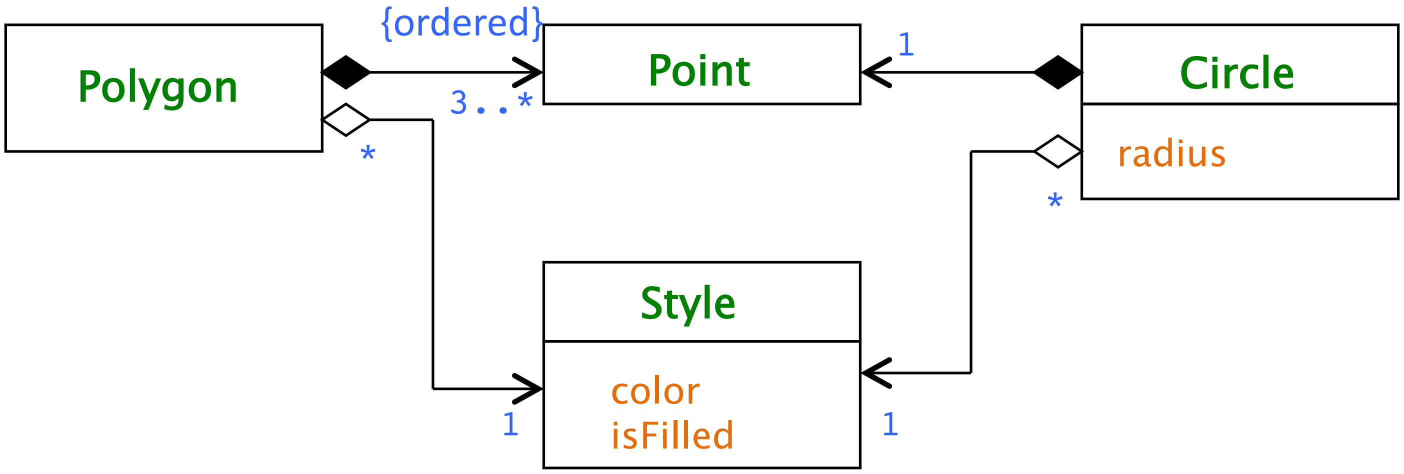
Examples of Aggregation

From “Object-Oriented Software Engineering” 2nd ed. by Bruegge and Dutoit



Example of Aggregation and Composition

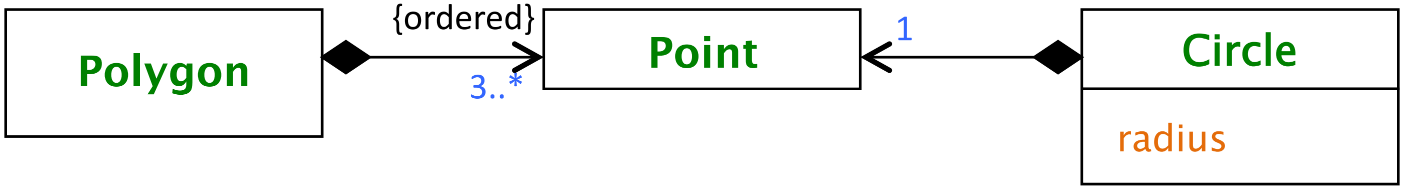
From “UML Distilled” By Martin Fowler with Kendall Scott



Point participates in two composition relationships: what could this mean?

All associations have a directional arrow: what could this mean?

Class participates in multiple compositions



```
public class Polygon {
   // implementation must
   // ensure at least 3 points
   // in sequence
   private Point[] points;
   ...
}
```

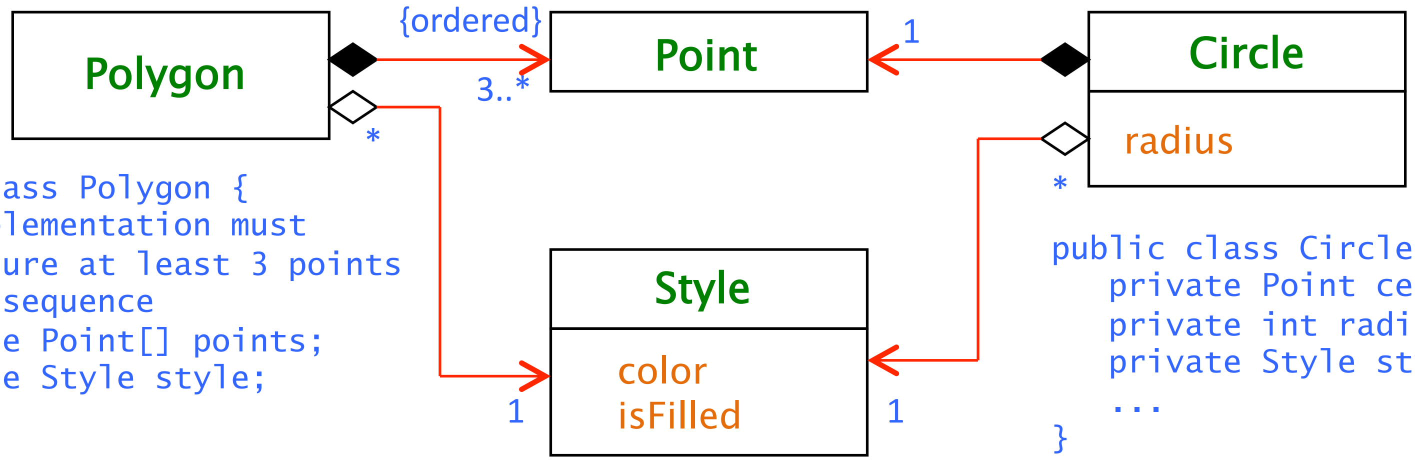
```
public class Point {
   private int x,y;
   ...
}
```

```
public class Circle {
   private Point center;
   private int radius;
   ...
}
```

Point participates in two compositions. The qualifier (design wise) is that the Point instance in the Circle is different from any of the Point instances in Polygon.
So that if a Circle instance, or Polygon instance, is no longer active, the contained Point instances can be safely destroyed.

However, in Java, which implements automatic garbage collection, this restriction does not apply. An instance of Point used in Circle can also be used in Polygon: if the Polygon instance goes out of scope, only the contained instances of Point that DO NOT have a reference from elsewhere will go out of scope as well. (If an instance is referred from a Circle instance, it will not be garbage collected.)

The meaning of directed associations



```
public class Polygon {
   // implementation must
   // ensure at least 3 points
   // in sequence
   private Point[] points;
   private Style style;
   ...
}
```

```
public class Circle {
   private Point center;
   private int radius;
   private Style style;
   ...
}
```

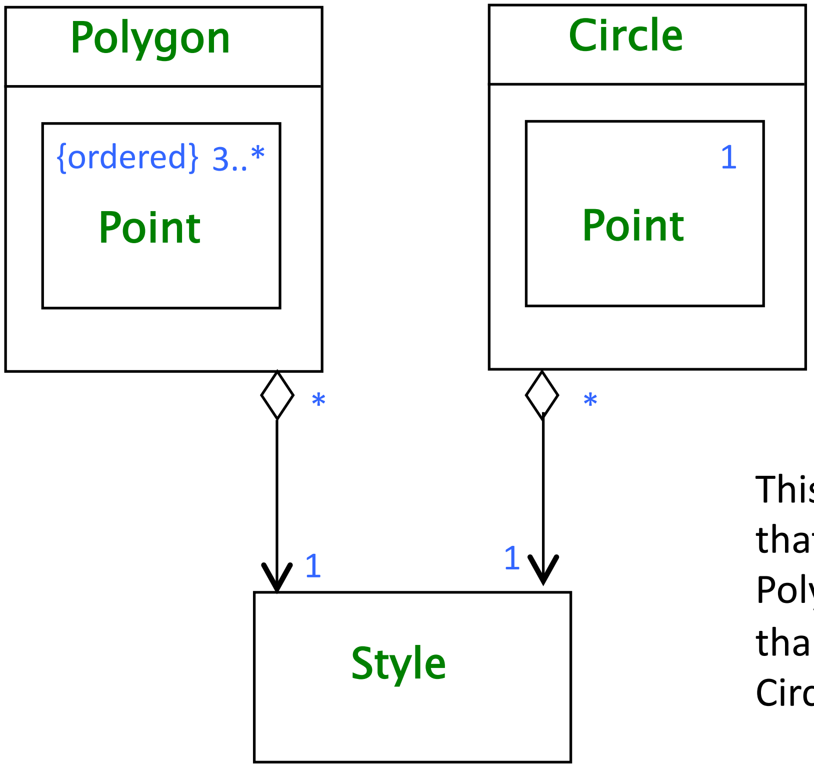
Polygon “knows” about its Style and Point associations (and so they are fields), but Style and Point do not know about their Polygon associations

```
public class Point {
   private int x,y;
   ...
   // NO REFERENCE TO
   // Polygon or Circle
}
```

```
public class Style {
   private Color color;
   private boolean isFilled;
   ...
   // NO REFERENCE TO
   // Polygon or Circle
}
```

Circle “knows” about its Style and Point associations (so, fields), but Style and Point do not know about their Circle associations

Alternative Notation for Composition



This notation makes it more obvious that the Point instances contained in Polygon are (design wise) different than the Point instances contained in Circle