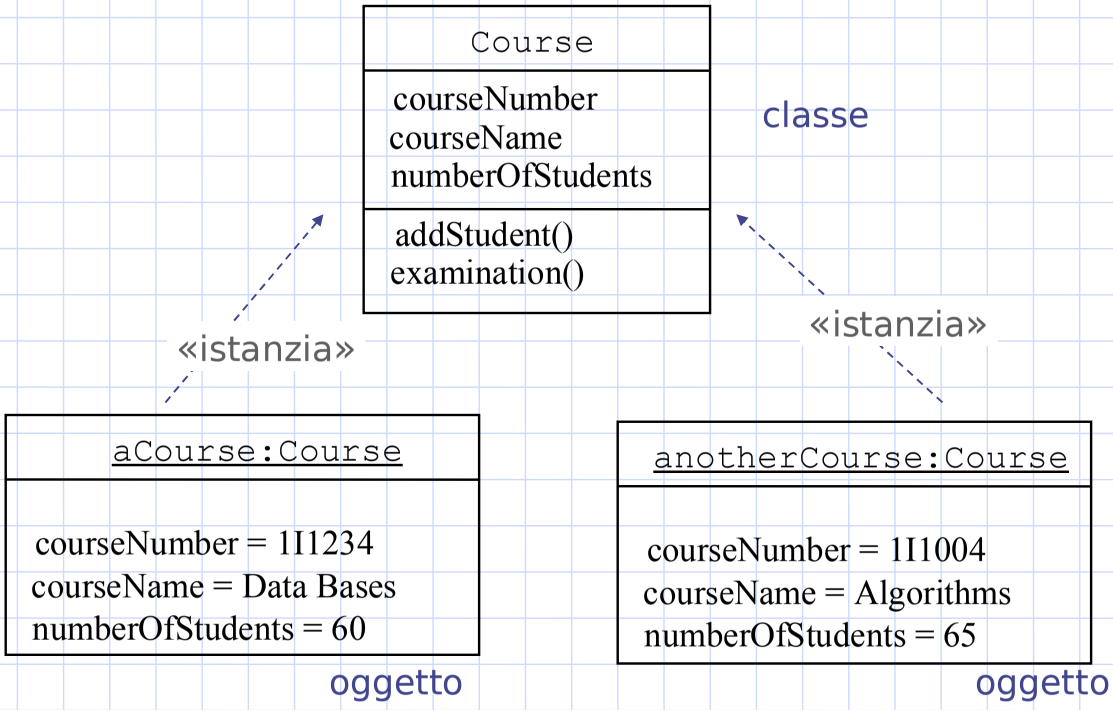
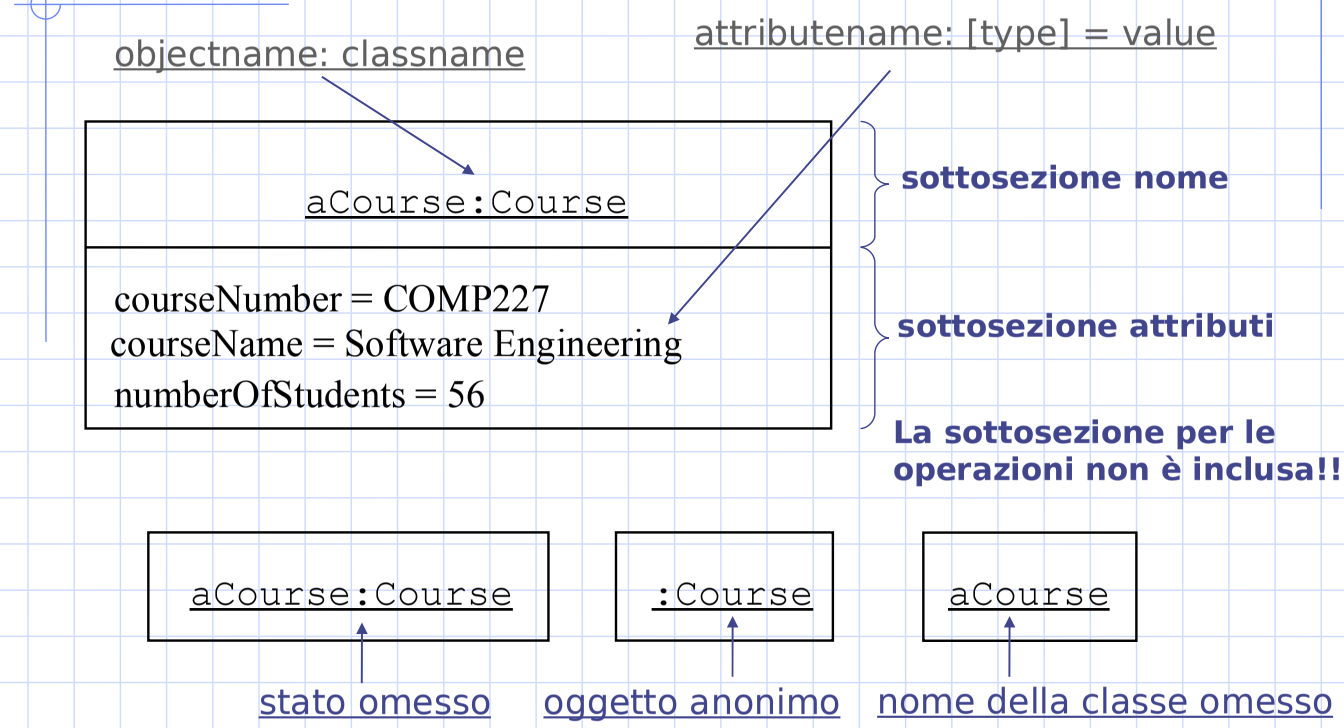
Relazioni tra Classi

1. Collegamenti tra oggetti
2. Relazioni tra classi
3. Dipendenza
4. Associazione
5. Composizione: Implementazione in C++ e Java
6. Aggregazione:Implementazione in C++ e Java
7. Altre relazioni tra classi

Classi ed oggetti



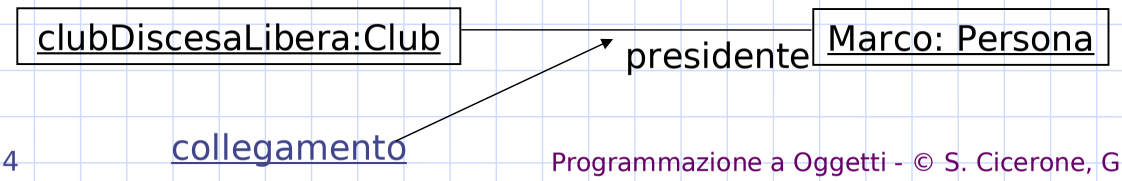
Notazione per gli oggetti



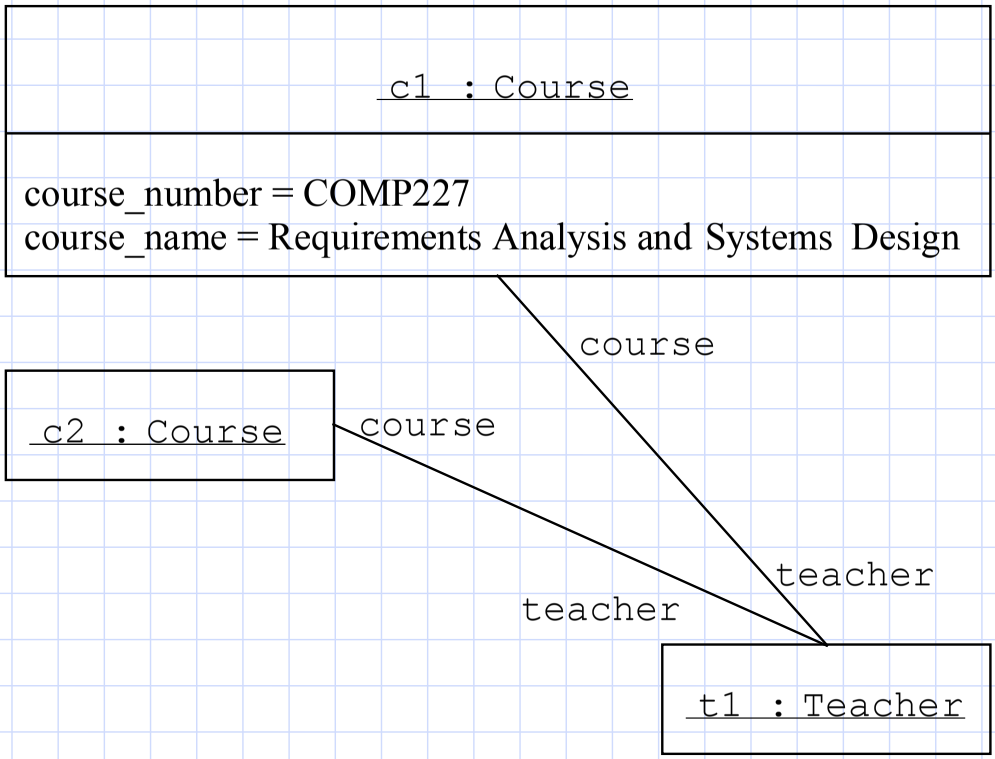
nome:classe sempre sottolineato

Cosa è un collegamento

- un programma OO richiede oggetti (fondamentali) che comunicano tra loro
- un **collegamento** tra 2 oggetti è una connessione semantica che consente agli oggetti di scambiarsi messaggi
- un programma OO in esecuzione prevede oggetti e collegamenti che nascono e muoiono
- I vari linguaggi OO realizzano i collegamenti in modi diversi
  - C++: puntatori, riferimenti, inclusione di oggetti interni
  - Java: riferimenti, inclusione di oggetti interni, *inner classes*



Collegamenti tra oggetti



Elenco relazioni tra classi

dipendenza

associazione

composizione

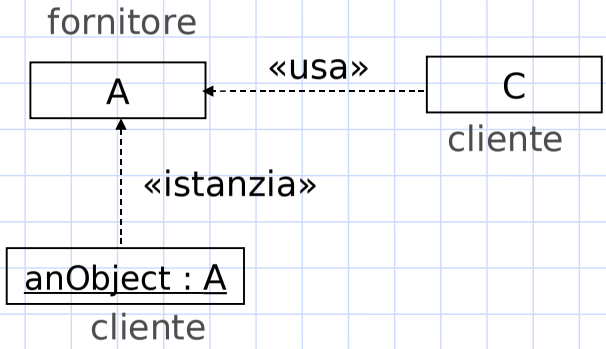
aggregazione

generalizzazione

realizzazione

Dipendenze

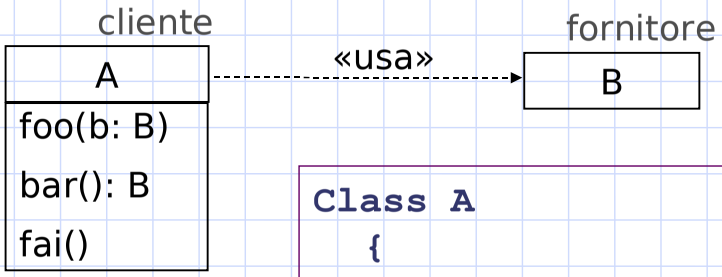
- “relazione tra due elementi, dove il cambiamento di uno di essi (fornitore) influenza o fornisce informazioni necessarie all’altro (cliente)”



  - relazione tra classi, e tra oggetti e classi

Dipendenze di uso

- dipendenza maggiormente usata tra le classi




```
Class A
   {                                        ...
     void fai() {
            B mioB = new B();
            //  usa mioB in qualche modo
            ...
      }
}
```

Associazioni

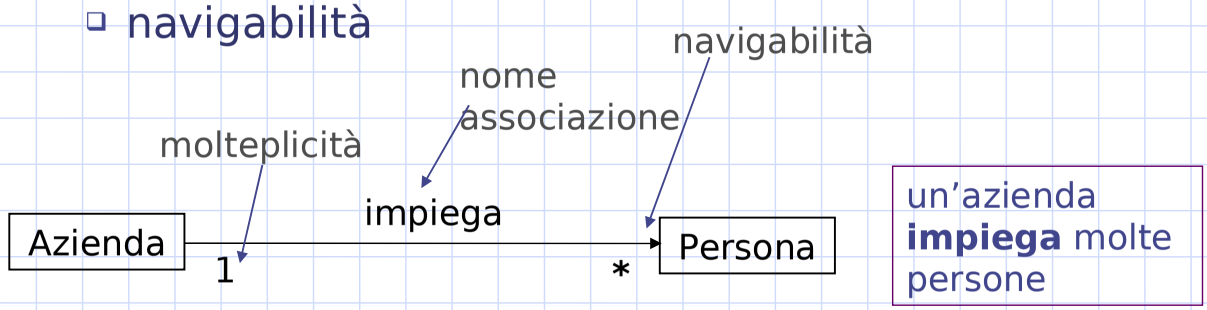
- Le associazioni sono relazioni tra classi
- Così come i collegamenti connettono gli oggetti, le associazioni connettono le classi.

> Se esiste un **collegamento tra oggetti**, allora deve esistere una **associazione tra le classi** corrispondenti

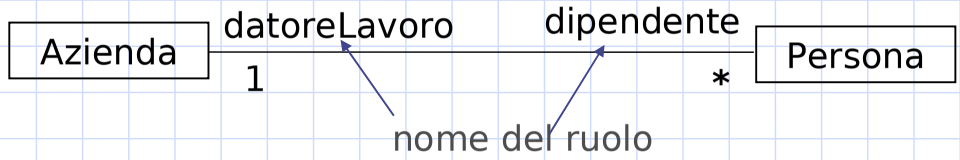
- Un collegamento è un'istanza di una associazione, così come un oggetto è un'istanzia di una classe.

Associazione: sintassi UML

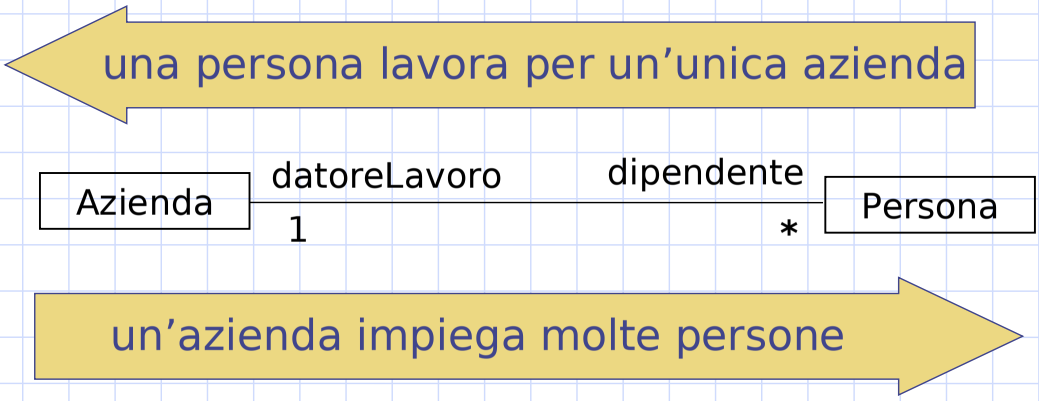
- le associazioni possono essere indicate con:
  - nome dell'associazione *(azione espressa tramite verbi o frasi verbali)*
  - nomi dei ruoli
  - molteplicità
  - navigabilità



un'azienda **impiega** molte persone



Molteplicità



La molteplicità riferita all'associazione esprime due regole di business:

- *gli oggetti Persona possono essere dipendenti di un'Azienda per volta*
- *gli oggetti Persona devono essere sempre dei dipendenti*

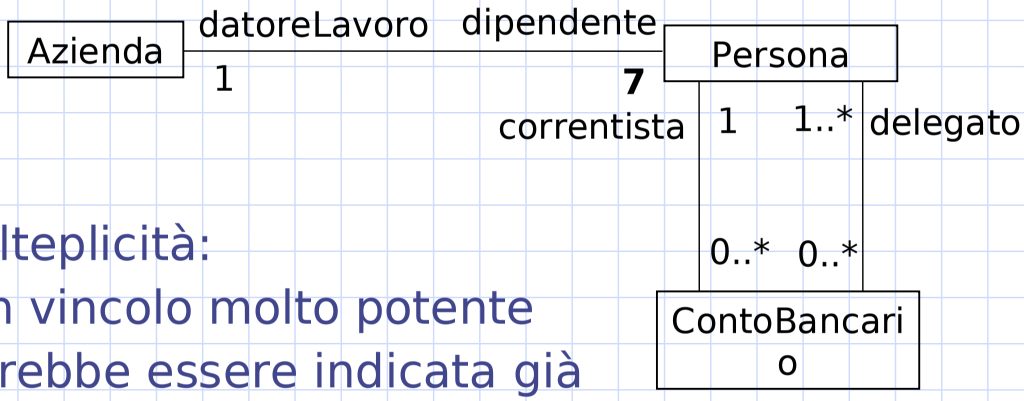
Molteplicità

La molteplicità è specificata da una lista di intervalli:

- 0..1 → zero o 1
- 1 → esattamente 1
- 0..* → zero o più
- * → zero o più
- 1..* → 1 o più
- 1..6 → da 1 a 6
- 1..3, 7..* → da 1 a 3 oppure 7 o più

se non è indicata allora è *indefinita* (errore!)

Molteplicità

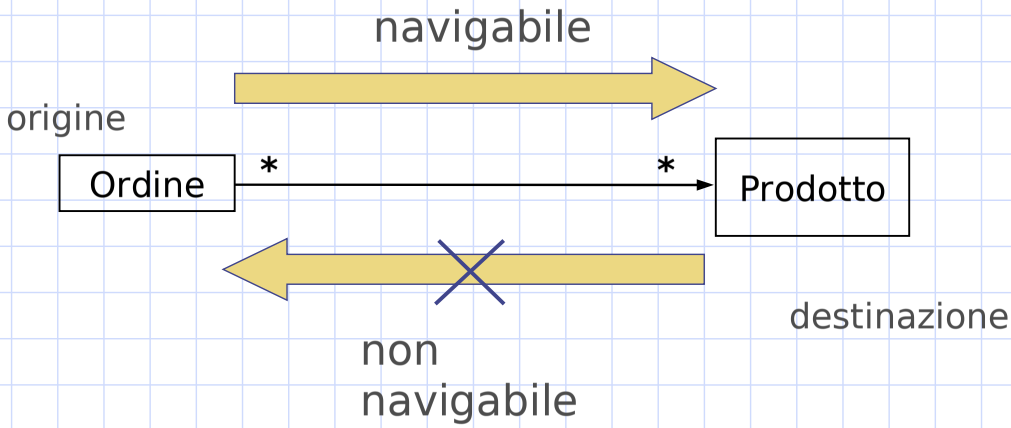


La molteplicità:

- è un vincolo molto potente
- dovrebbe essere indicata già dai modelli di analisi (esprime regole di business)

Navigabilità

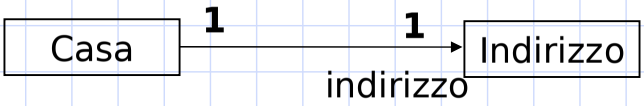
- modo per ridurre l'interdipendenza tra classi !
- collegamenti senza frecce = navigabilità bidirezionale



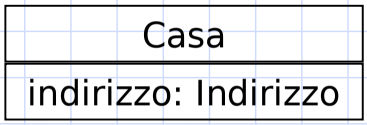
- gli oggetti Prodotto non sanno di essere in relazione con oggetti Ordine

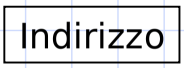
Associazioni ed attributi

- realizzazione di associazioni 1 a 1:



modello di analisi





modello di analisi:
il nome del ruolo genera un attributo

```
public class Casa {

  private Indirizzo indirizzo;

}
```

codice Java per la classe Casa
(es., generato da un tool CASE)

Associazioni ed attributi

- Le molteplicità superiori ad 1 si realizzano come:
  - attributo di tipo vettore *(es., java.util.Vector)*
  - attributo di tipo contenitore *(es., list<> di STL in C++)*
- *Vettori:*
  - insieme indicizzato
  - veloci
  - poco flessibili
- *Contenitori:*
  - specializzati per permettere aggiunte, rimozioni, ricerche di oggetti generici
  - eccellenza nella progettazione → padronanza nell'uso delle classi contenitore

Associazioni ed attributi

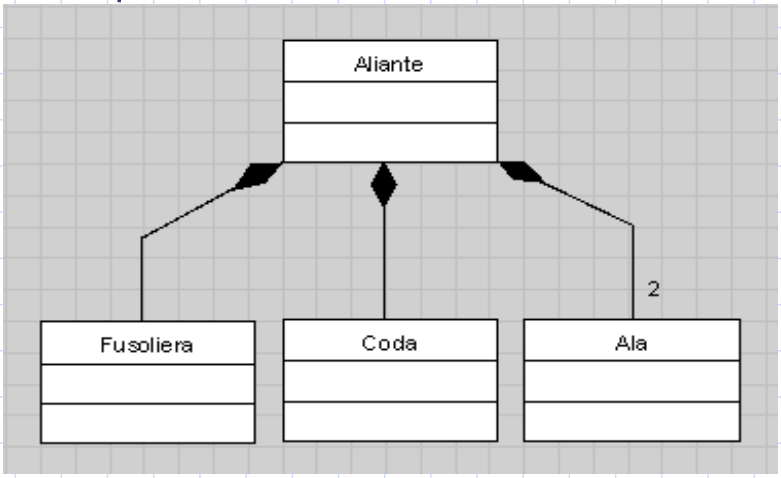
- Fase di analisi:
  - si definiscono le classi Entity – le classi che modellano gli oggetti fondamentali del dominio applicativo (oggetti di business)
  - gli attributi nelle classi Entity sono sempre di tipo primitivo
  - l'uso di tipi non primitivi implica la necessità di definire associazioni tra le classi Entity *(attenzione a navigabilità e molteplicità)*
- Fase di design:
  - le associazioni possono essere raffinate aggiungendo ulteriore semantica: *aggregazioni, composizioni* (vedi)
  - le associazioni vengono realizzate in modi diversi a seconda della molteplicità

Composizione/Aggregazione

- Quando in una classe si includono oggetti appartenenti ad altre classi si realizza il concetto di "*associazione di tipo tutto/parte tra classi*"
- Due sono i tipici modi di realizzare questa associazione: **composizione** e **aggregazione**. Sono raffinamenti dell'associazione.
- Nel primo caso l'oggetto interno esiste solo per poter realizzare la "parte" dell'oggetto esterno
- Nel secondo caso l'oggetto interno esiste indipendentemente del suo ruolo di "parte".

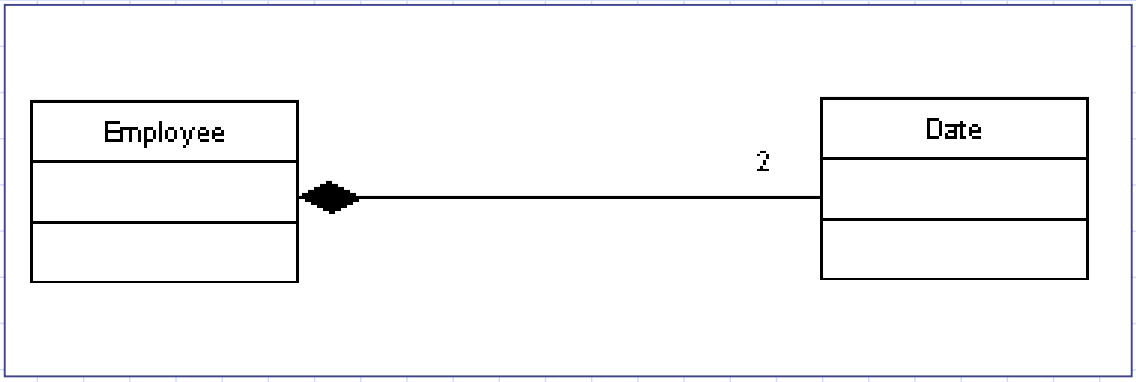
Classe: composizione

- Il “tutto” = **composto**, la “parte” = **componente**
- Tre le caratteristiche più importanti:
  - L’oggetto composto non esiste senza i suoi componenti
  - In qualunque momento, ciascun oggetto componente dato può essere parte di un solo composto
  - La composizione tipicamente è eterogenea (i componenti sono di tipi misti)



Composizione in C++ (molteplicità finita)

- Costruzione di nuove classi mediante inclusione di oggetti appartenenti a classi già definite
- L’oggetto contenitore è responsabile del ciclo di vita dell’oggetto contenuto
- Il costruttore dell’oggetto contenitore deve chiamare il costruttore dell’oggetto contenuto nella lista di inizializzazione



Employee.h (interfaccia)

```
#include <string>
#include "Date.h"

class Employee {
public:
Employee( string, string, int, int, int, int, int, int );
   void print() const;    // stampa
   ~Employee();           // per confermare la distruzione
private:
   string firstName;
   string lastName;
   Date birthDate;   // uso di un oggetto esterno
   Date hireDate;    // uso di un oggetto esterno
};
```


Employee.cpp (parte dell'implementazione)

```
Employee::Employee( string fname, string lname,
                    int bmonth, int bday, int byear,
                    int hmonth, int hday, int hyear )
  : birthDate( bmonth, bday, byear ),  // lista di inizializzazione
    hireDate( hmonth, hday, hyear )    // uso costruttore in Date
{
  firstName = fname;// costruisco solo la parte propria di Employee
  lastName = lname; // ...
}

void Employee::print() const {
   cout << lastName << ", " << firstName << "\nHired: ";
   hireDate.print();          // uso i metodi in Date per la parte interna
   cout << "  Birth date: ";
   birthDate.print();         // uso i metodi in Date per la parte interna
   cout << endl;
}
```

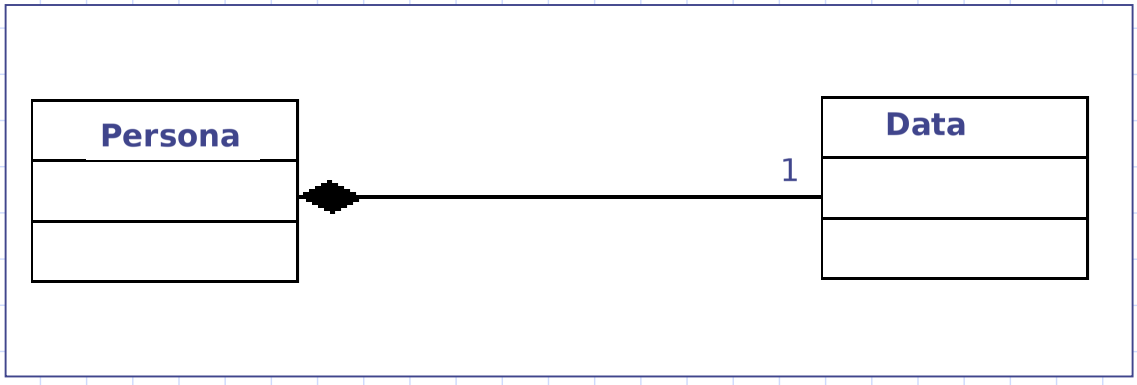
Suggerimento

Studiare l'esempio Employee<->Date poiché utilizza una classe Date che:

- È abbastanza completa (può essere utilizzata per i progetti)
- Contiene esempi completi di overloading di operatori
- Contiene metodi che utilizzano tutti e tre i modi per ritornare oggetti (per valore, per riferimento, per riferimento costante)

Composizione in Java (molteplicità finita)

- Valgono le stesse considerazioni dell'implementazione in C++
- L'oggetto contenitore è responsabile del ciclo di vita dell'oggetto contenuto
- Al contrario del C++, il costruttore dell'oggetto contenitore deve creare l'oggetto contenuto



Class Data in Java

```
public class Data {
      int giorno;
      int mese;
      int anno;

      public Data( int g, int m, int a){
          giorno =g;
          mese = m;
          anno = a;
      }

...

}
```


Class Persona in Java

```
public class Persona {
    private String nome;
    private String cognome;
    private Data dataNascita;

    public Persona(String n, String c, Data d) {
        nome= n;
        cognome = c;
        dataNascita = d;  //errore: d è solo un riferimento
                          //così come dataNascita
        //dataNascita = new Data(d.giorno,d.mese,d.anno);

    }

    public String getNome(){return nome;}
    public String getCognome(){ return cognome;}
    public Data getData(){return dataNascita;}
}
```

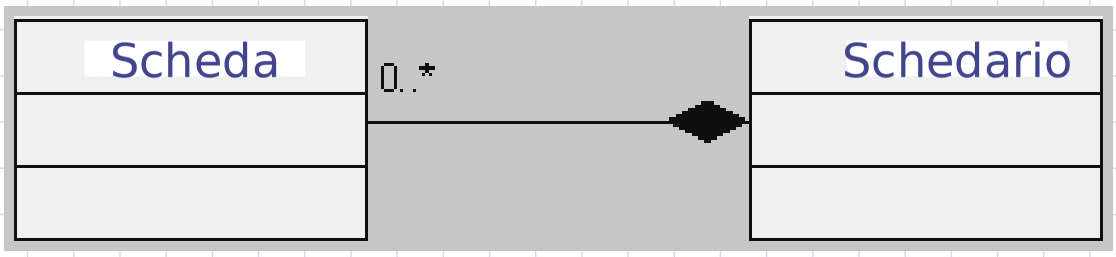
Approfondimento

L'esempio Persona<->Data utilizza una classe Data:

- Persona ha un attributo della classe Data;
- Ogni attributo di un tipo non primitivo è un riferimento ad un oggetto;
- Se si vuole che la classe Persona usi un attributo di classe Data come **componente** deve costruire l'oggetto relativo di classe Data.
- I componenti non vanno restituiti da metodi tipo get

Composizione in C++: molteplicità illimitata

- Costruzione di nuove classi mediante inclusione di oggetti appartenenti a classi già definite
- L'oggetto contenitore è responsabile del ciclo di vita dell'oggetto contenuto
- Metodi opportuni richiedono dati con cui costruiscono oggetti che poi rendono parte del tutto
- Scheda<->Schedario: L'oggetto contenitore è lo schedario, le schede sono gli oggetti contenuti (vedi codice dell'esempio*ComposizioneIllimitata*)



Discussione

Comprendere perché nell'esempio Scheda<->Schedario:

- la classe Scheda contiene l'overloading di:
  - operator<
- la presenza del precedente operatore rende la classe Scheda "nice" rispetto alla STL
- In questo modo, la classe Scheda può essere utilizzata nei contenitori che mantengono gli elementi ordinati come set e map.

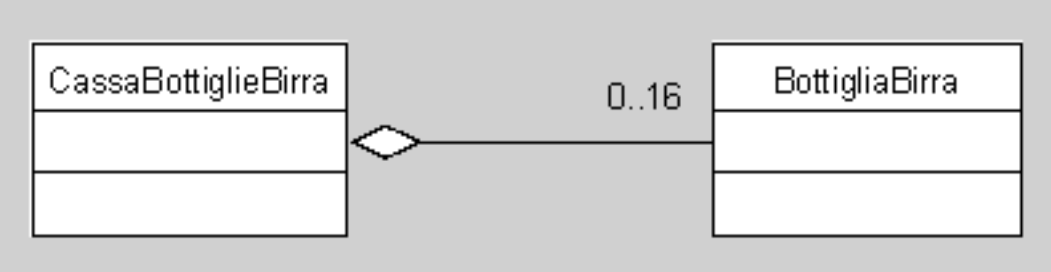
Esercizio in Java di composizione illimitata

Realizzare in Java l'esempio Scheda<->Schedario:

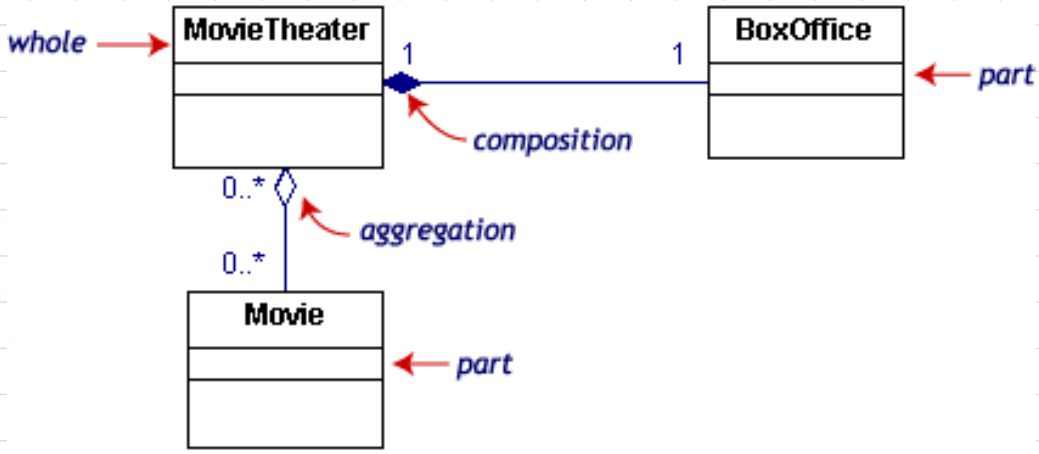
- La classe Schedario deve avere un **contenitore** adatto in cui inserire oggetti di tipo Scheda
- Per ogni scheda da inserire deve essere fornito in Schedario un metodo add() con opportuni parametri
- Il metodo add() deve costruire un oggetto scheda e inserire il riferimento nel contenitore
- In questo modo, nessun metodo esterno alla classe Schedario può modificare le schede, a meno di non fornire esplicitamente un metodo di accesso alle schede.

Aggregazione

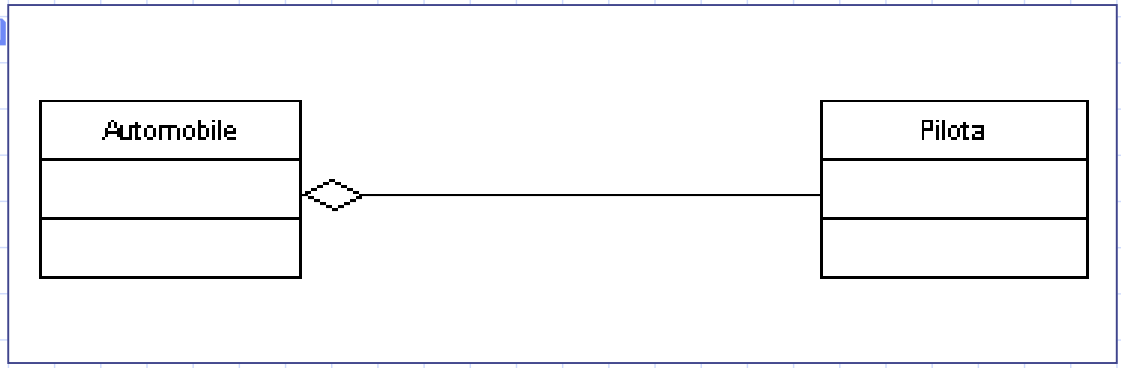
- Il “tutto” = **aggregato**, la “parte” = **costituente**
- Tre le caratteristiche più importanti:
  - L’oggetto aggregato potenzialmente può esistere senza i suoi oggetti costituenti
  - In qualunque momento, ciascun oggetto può essere costituente di più di un aggregato
  - L’aggregazione tende a essere omogenea (i componenti sono della stessa classe)



Esempio

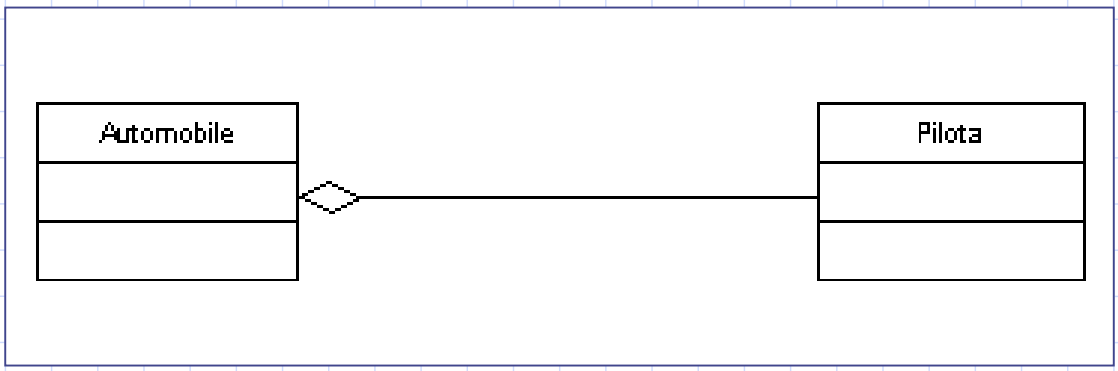


Aggregazione in C++

- Costruzione di nuove classi mediante inclusione di oggetti appartenenti a classi già definite
- L'oggetto contenitore non è responsabile del ciclo di vita dell'oggetto contenuto
- Implementazione effettuata tramite l'uso di puntatori
- 

Aggregazione in Java

- Stesse considerazioni del C++
- L'oggetto contenitore non è responsabile del ciclo di vita dell'oggetto contenuto
- Implementazione effettuata tramite l'uso di riferimenti (naturale in Java)



Esempio riassuntivo