OOP
Relazioni tra classi

Derivazione ed ereditarietà

Associazione

Aggregazione composizione

Derivazione ed ereditarietà

Derivazione

Richiamiamo il concetto di derivazione.
Quando si deriva una classe da un'altra:

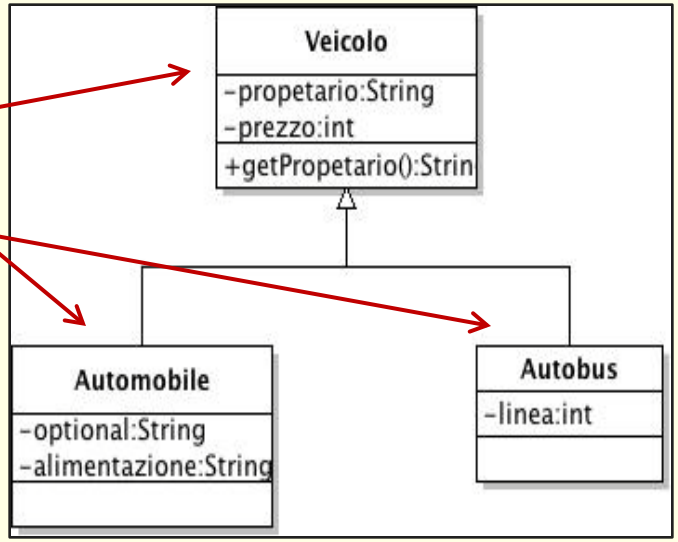
> la classe derivata
> **eredita**
> **tutti gli attributi** e **tutti i metodi**
> definiti *pubblici* o *protetti* (ma *non quelli privati*)
> appartenenti alla classe padre.

Una **classe derivata** può essere considerata un'**estensione** di una classe oppure una **classe che eredita** le proprietà e i metodi da un'altra classe.
Consente di espandere o personalizzare le funzionalità di una classe base, senza costringere a modificare la classe base stessa.

La ***classe originaria*** viene denominata **classe base**.

La ***classe derivata*** viene anche chiamata **sottoclasse**.

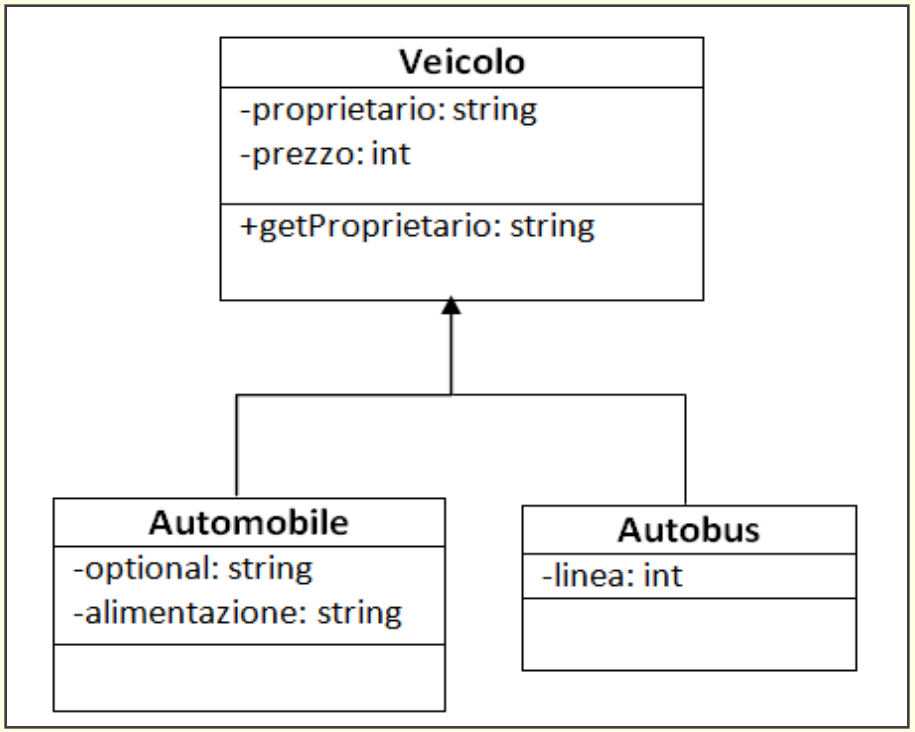
E' possibile **derivare più classi da una singola classe base.**

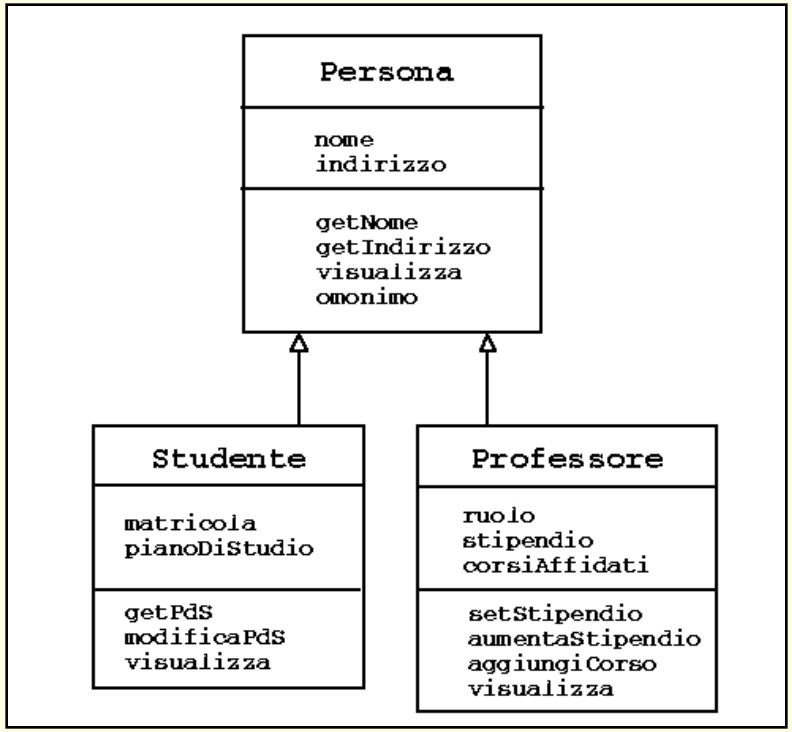


Derivazione ed ereditarietà

La **derivazione** è spesso utilizzata per **specializzare** una classe che possiede attributi più generali.
In UML il concetto di ereditarietà si rappresenta con una linea con una freccia con una punta a triangolo in direzione della classe padre.

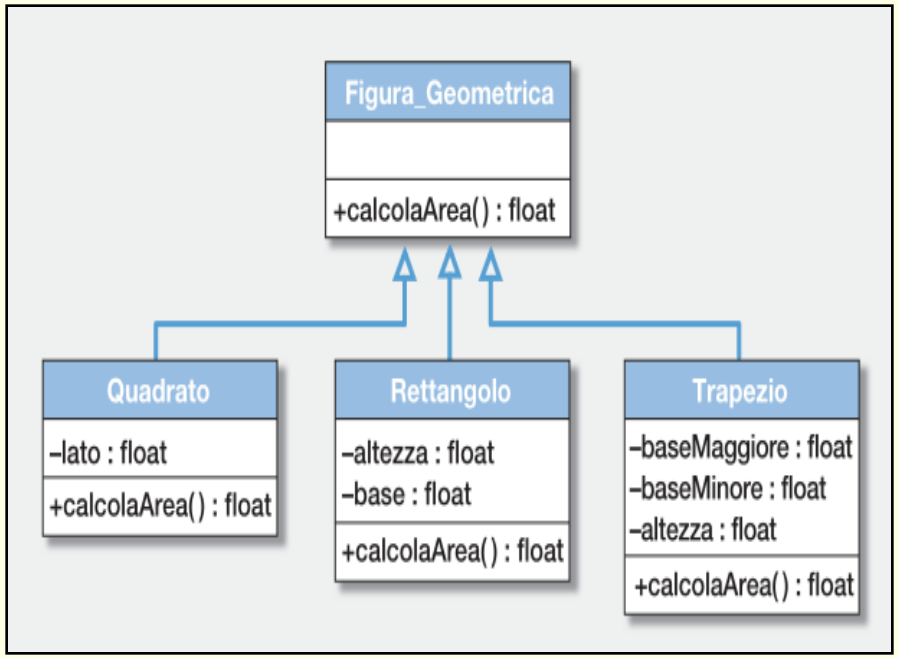
Concetto di specializzazione: la classe Autobus è più ricca di informazioni rispetto alla classe padre Veicolo e quindi risulta più specifica.



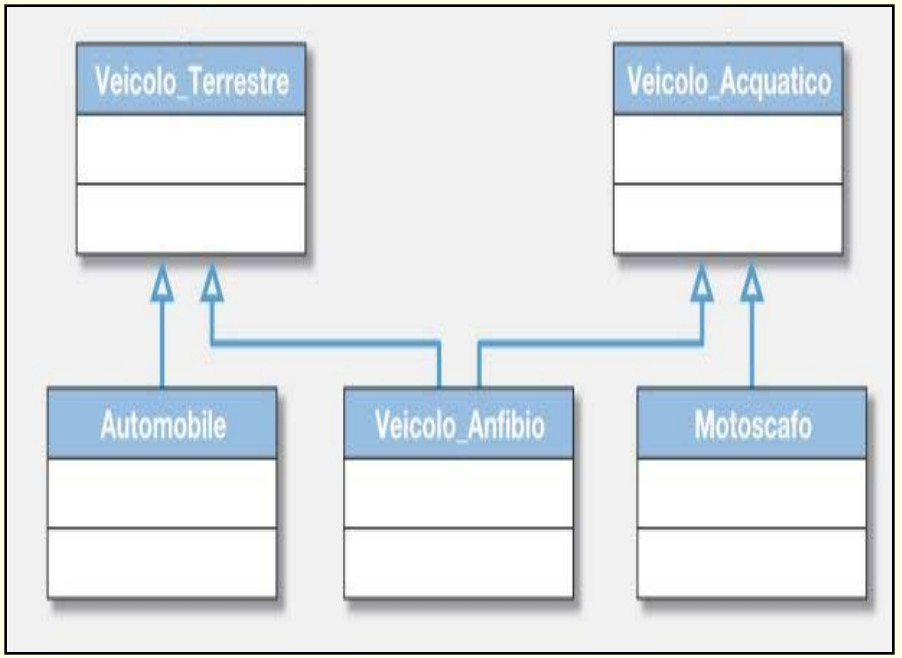


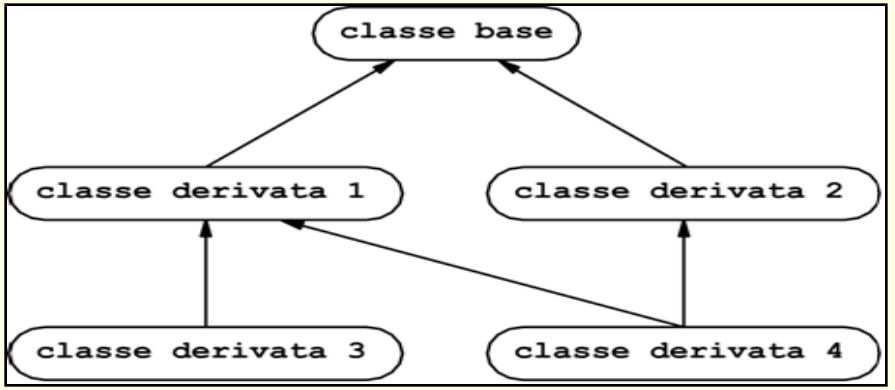
Derivazione ed ereditarietà

Esempio Derivazione Singola



Esempio Derivazione Multipla





Derivazione ed ereditarietà

La **classe derivata** può:

- **Ereditare** attributi e metodi della classe base
- **Aggiungere** nuove funzionalità alla classe base
  - **modificare i privilegi** d'accesso
  - aggiungere **nuove funzioni membro** e **nuovi attributi**
  - **modificare tramite overloading** le funzioni membro esistenti
  - **riscrivere tramite overriding** le funzioni membro esistenti

Sintassi di una classe derivata

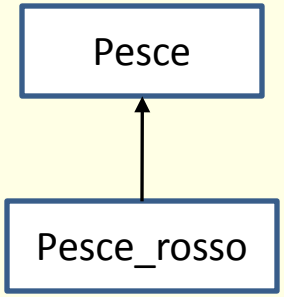
Per descrivere una classe derivata si fa uso della seguente sintassi:

```
class classe derivata : <specificatore d'accesso> classe base
  {
  ......
  ......
  };
```

Esempio

```
class Pesce_rosso : public Pesce
{
  ......
  ......
};
```

La **classe derivata** si chiama pesce_rosso.
La **classe base** ha visibilità pubblica e si chiama pesce.



Derivazione ed ereditarietà

Il meccanismo di ereditarietà, pur essendo abbastanza semplice, richiede una certa attenzione, perchè dipende dallo standard della classe base.

Gli **attibuti** ed i **membri**

che vengono **ereditati** dalla ***classe base***

possono cambiare la loro visibilità nella ***classe figlia***,

in base allo **specificatore d'accesso** con il quale si esegue l'ereditarietà stessa.

```
class   classe derivata :   <specificatore d'accesso>   classe base
  {
  ......
  ......
  };
```

Lo **specificatore d'accesso** indica il tipo di eredità e può essere: ***public***
protected
private

Derivazione ed ereditarietà

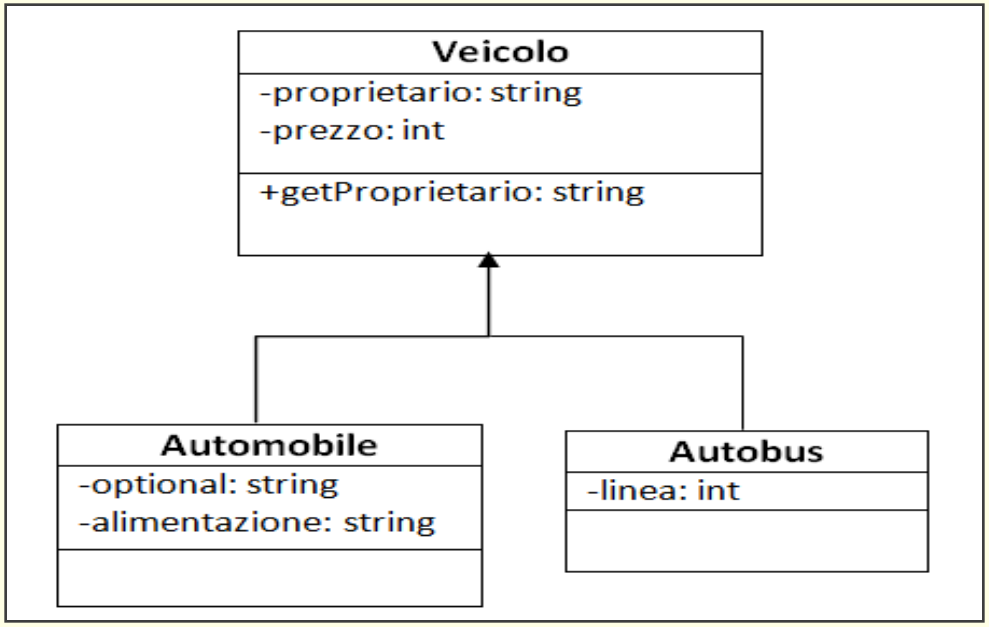
Regole

Con l'ereditarietà un accesso può restringersi o restare uguale ma mai ampliarsi.

Membri della classe base o superclasse	diventano nella classe derivata definita con specificatore *public*	diventano nella classe derivata definita con specificatore *protected*	diventano nella classe derivata definita con specificatore *private*
public	public	protected	private
protected	protected	protected	private
private	non accessibili	non accessibili	non accessibili

```
class  classe derivata :  <specificatore d'accesso>  classe base
  {
  ......
  ......
  };
```

Derivazione ed ereditarietà



Quando verrà
Istanziato un oggetto
della classe Autobus

automaticamente

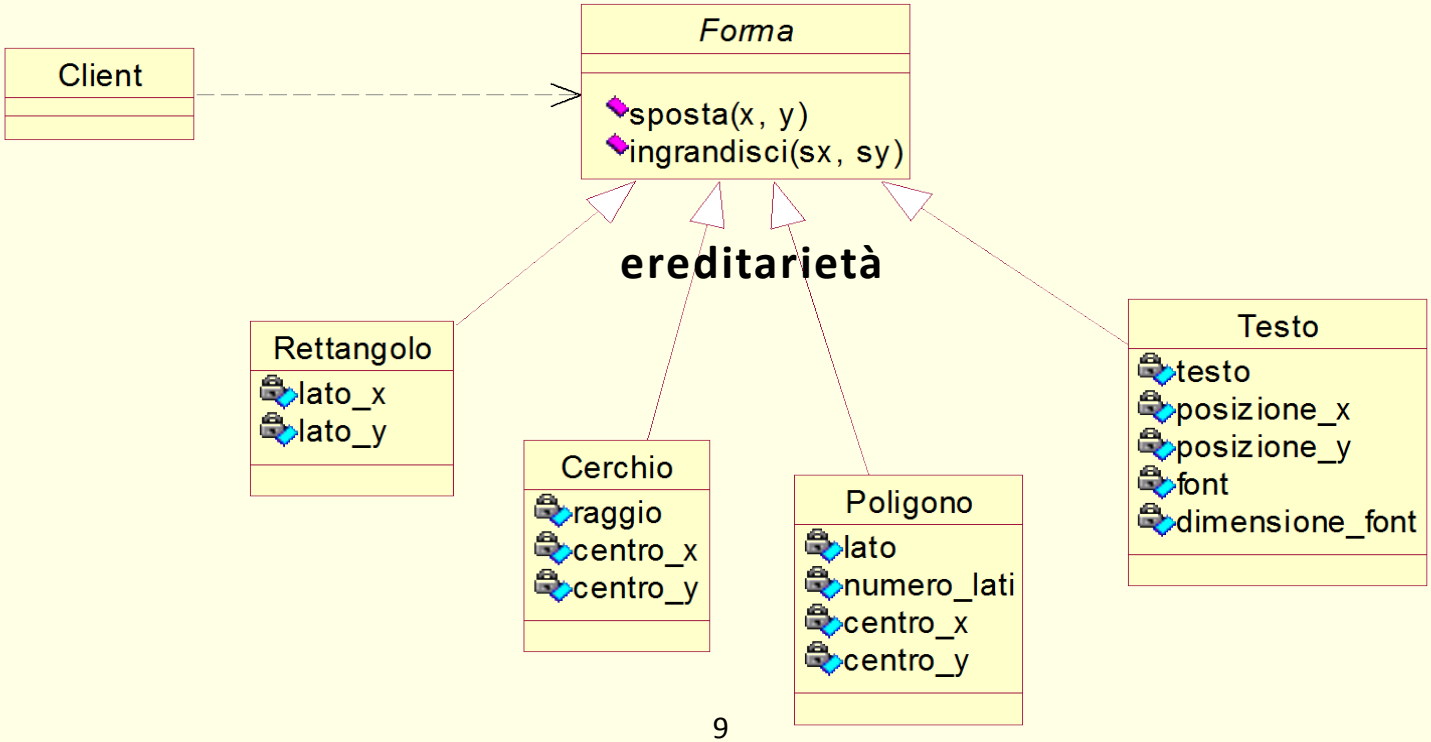
verrà
richiamato il costruttore
della classe Veicolo
che inizializzerà
i due attributi della classe

Se si vuole conoscere il proprietario dell'Autobus, non sarà possibile accedere direttamente all'attributo, in quanto definito privato, ma si potrà utilizzare il metodo getProprietario() (definito pubblico) che restituirà il valore dell'attributo proprietario.

Derivazione ed ereditarietà

Esempio

- I messaggi comuni (sposta, ingrandisci, …) sono definiti in maniera astratta nella **classe di base "**Forma"
- Le **sottoclassi** li **ereditano** e li **implementano** in modo specifico
- Il programma "client" ignora i dettagli di implementazione e gestisce in modo omogeneo tutti i sottotipi di "Forma"



Derivazione ed ereditarietà

Esempio

Il metodo calcolaArea della classe Figura_Geometrica è ridefinito in tutte le classi derivate: questo significa che ogni classe implementa uno specifico algoritmo di calcolo dell'area mantenendo solo la firma del metodo ereditato.

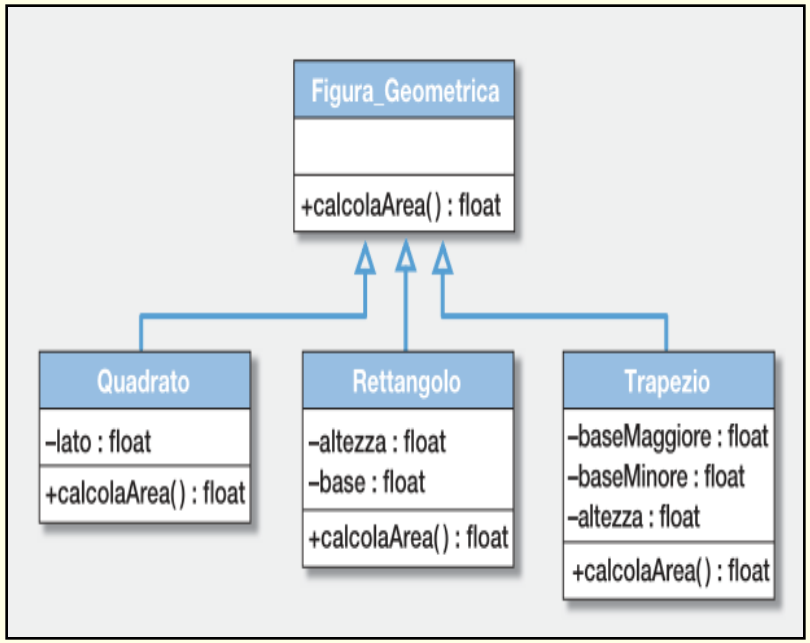
Per calcolare l'area di una figura geometrica disponendo di:

- un **oggetto *figura*** istanza di una qualsiasi classe della gerarchia

è quindi sufficiente invocare questo metodo:

```
...
area = figura.calcolaArea();
...
```

```
int main()
{Quadrato figura1;
  Trapezio figura2;
   float area1,area2;
   ...
  area1 = figura1.calcolaArea();
  area2 = figura2.calcolaArea();
...
```

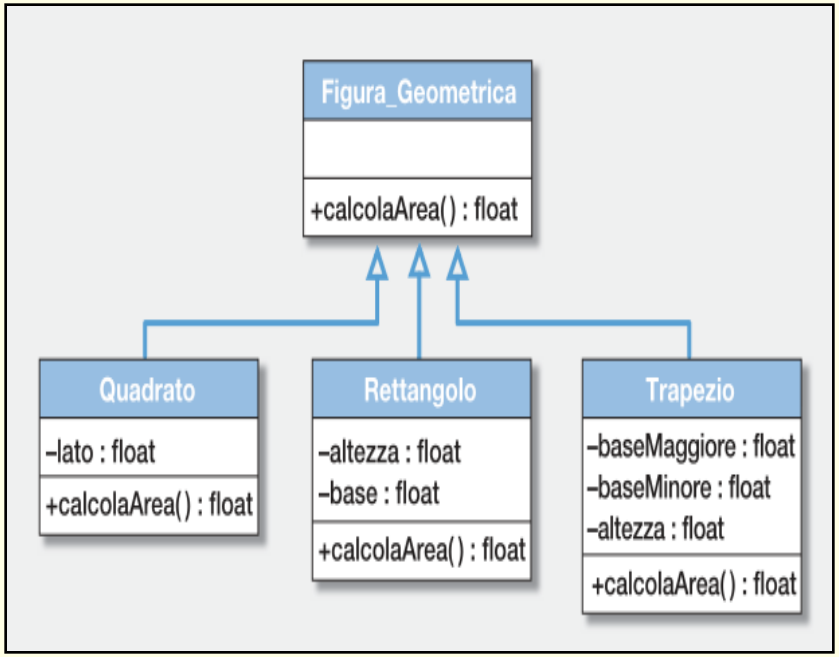


Applicando il **polimorfismo** il codice è in grado di determinare il metodo da invocare (quello definito nella classe Quadrato, oppure nella classe Rettangolo, o nella classe Trapezio) in funzione della classe di cui l'oggetto figura è istanza.

Derivazione ed ereditarietà

Utilizzando un linguaggio di programmazione non orientato agli oggetti, e quindi privo del polimorfismo, il codice sarebbe invece risultato simile al seguente:

```
…
if (nomeFigura == “Quadrato”)
     area = calcolaAreaQuadrato();
else
    {if (nomeFigura == “Rettangolo”)
        area = calcolaAreaRettangolo();
      else
        area = calcolaAreaTrapezio();
    }
…
```



Relazione di associazione

E possibile legare varie classi presenti in un progetto con una **relazione di associazione**.

Una **associazione** individua una ”connessione” logica tra classi quindi tra le istanze delle classi coinvolte

In UML viene rappresentata da una linea che collega due classi.

- agli *estremi della linea* si indicano le **molteplicità della relazione** (cardinalità), ovvero la quantità di oggetti di una classe che sono legati all'altra
- al *centro* della linea viene espresso con un verbo cosa indica la relazione.

Esempi di molteplicità
0..1(da zero ad una istanza)
0..* (zero o più istanze)
1..* (una o più istanze)
1..6 (da una a sei istanze)

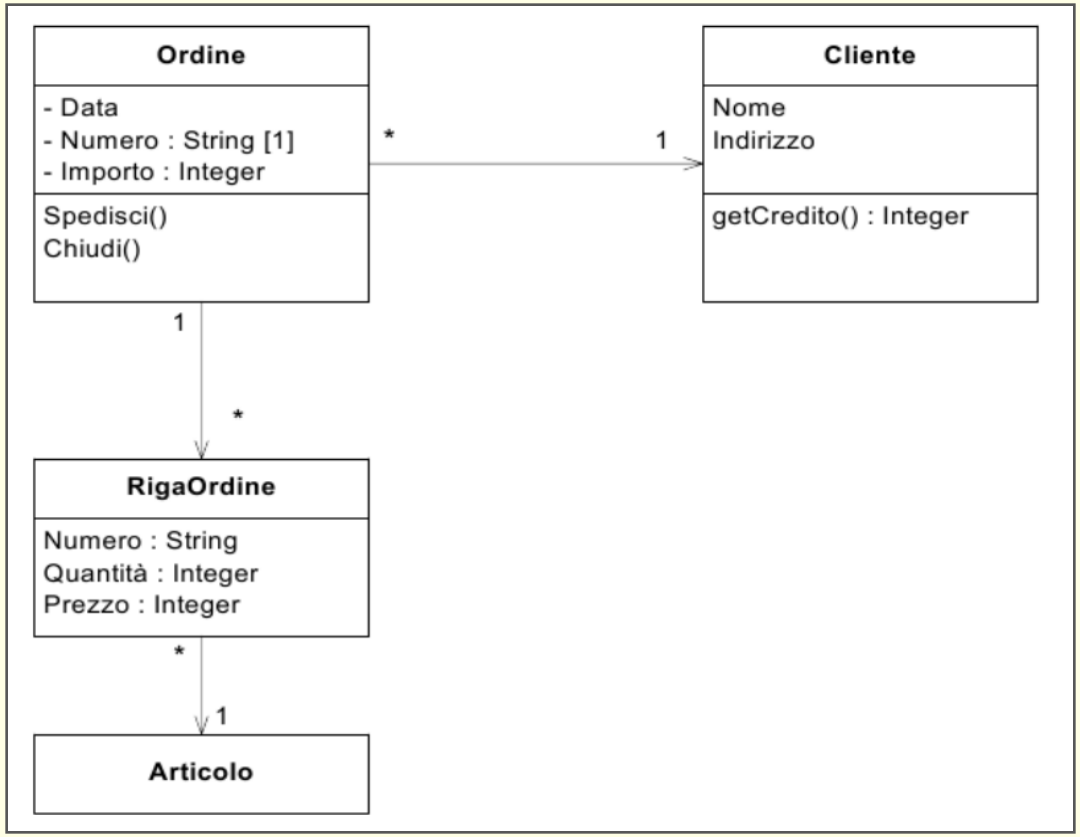
Esempio

Associazioni:

- La prima relazione tra Cuoco e Ristorante indica:
  - *Un cuoco lavora in un ristorante*
  - *Un ristorante ha uno o più cuochi*
- Nella seconda è presente una freccia che indica la navigabilità dell'associazione e stabilisce un tipo di associazione monodirezionale.
  - *Un ristorante ha uno o più proprietari* e non si ha nessuna informazione riguardo alla relazione che intercorre tra Proprietario e Ristorante.

Relazione di associazione

L'associazione è una relazione che collega classi costruite in modo indipendente.



Le associazioni sono di 3 tipi:

- **uno-a-uno**
- **uno-a-molti**
- **molti-a-molti**

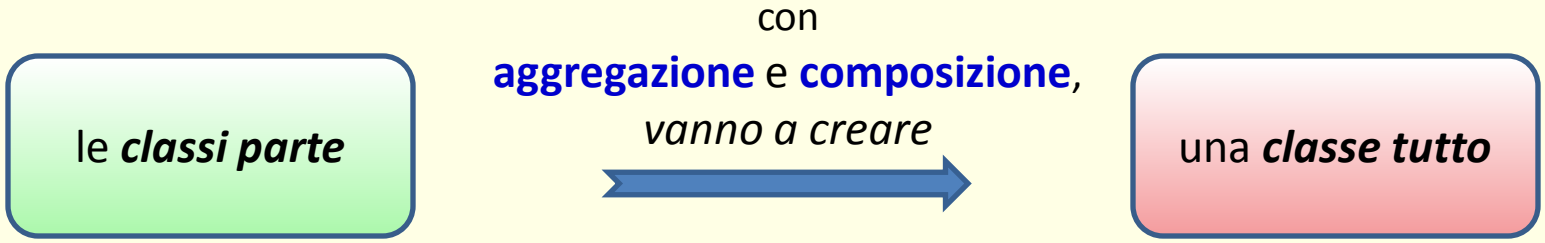
Nell'esempio:

- la classe **Ordine** contiene un puntatore che fa riferimento alla classe **Cliente**.
- la classe **Ordine** contiene un array che fa riferimento a diverse **RigaOrdine**
- la classe **RigaOrdine** contiene un puntatore che fa riferimento ad **Ordine** e un puntatore ad **Articolo.**

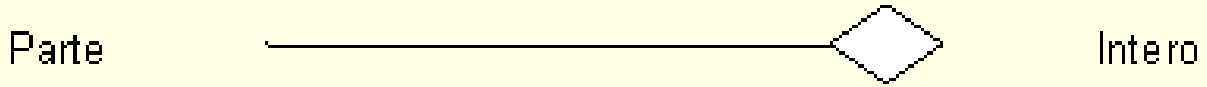
Aggregazione e Composizione

Aggregazione e **composizione**

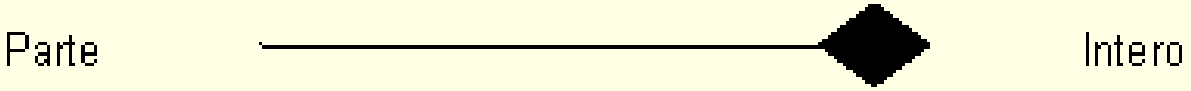
Sono speciali forme di associazione che specificano una relazione ***whole-part*** *(tutto-parte)* tra l'aggregato (contenitore) e le parti componenti



- **Aggregazione**: relazione **non forte** (le **parti esistono** anche **senza il tutto**).
  Le *classi parte* ***hanno un significato*** **anche** ***senza che sia presente*** *la classe tutto*.

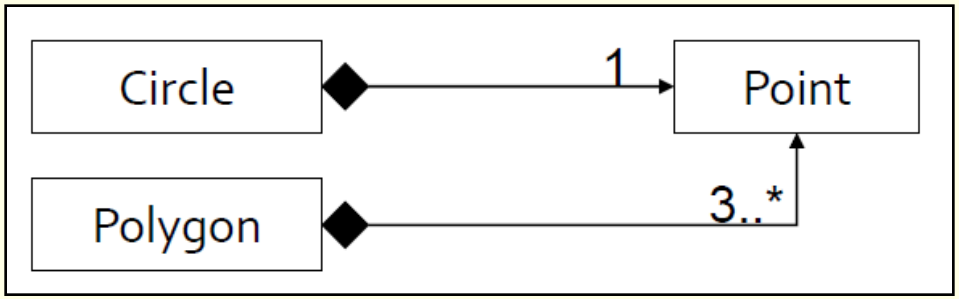


- **Composizione**: relazione **forte** (le **parti dipendono** dal **tutto**, per esempio le pareti e la stanza).
  Le *classi parte* ***hanno un reale significato solo se*** *sono legate alla classe tutto* quindi se l'aggregato viene distrutto anche le sue parti vengono distrutte.

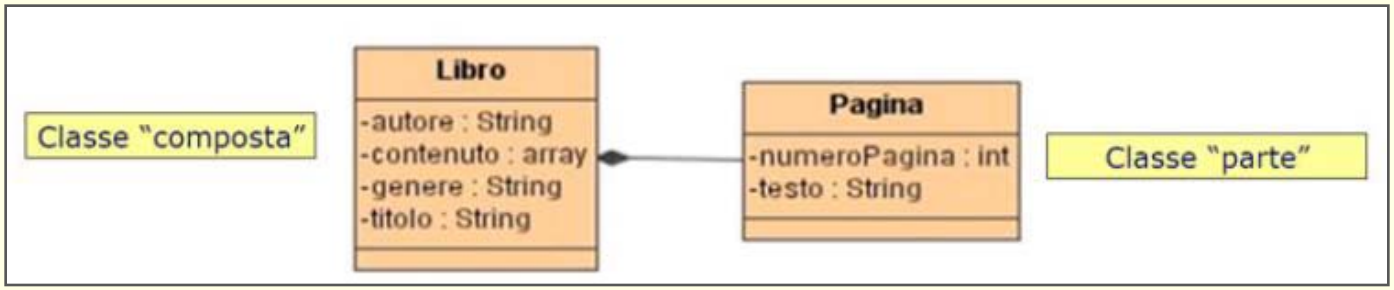


Composizione

- La composizione viene considerata una **forma forte** di aggregazione
- Il "**tutto**" **è il solo proprietario** (owner) delle proprie parti, dette "componenti"
  - L'oggetto "**parte**" appartiene al più ad un unico "**tutto**"
  - La **molteplicità** dalla parte del "tutto" può essere solo 0 oppure 1

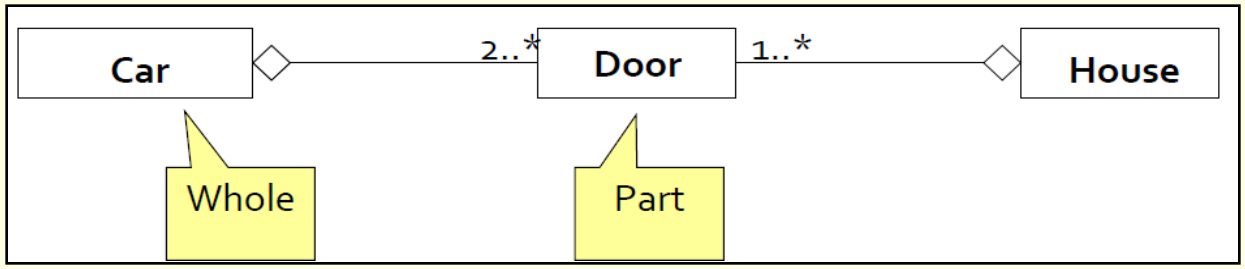


- Il **tempo di vita** (life-time) della "parte" è subordinato (minore o uguale) a quello del "tutto"; quando l'oggetto "tutto" viene distrutto non esiste più neppure l'oggetto "parte"
  - L'oggetto "**tutto**" gestisce la **creazione** e la **distruzione delle sue parti**

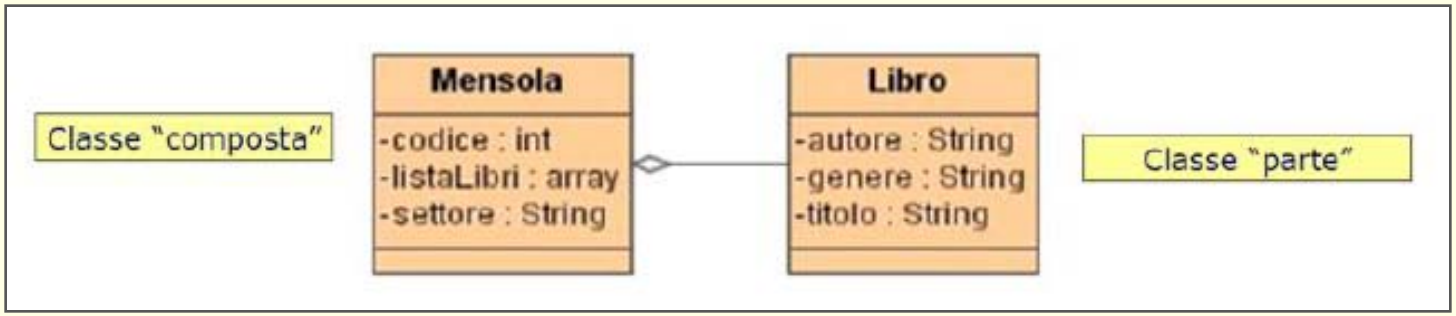


Aggregazione

- Il "**tutto**" **NON è il solo proprietario** (owner) delle proprie parti
  - L'oggetto "**parte**" può essere condivisa da più oggetti "**tutto**"
  - Non ci sono limiti alla **molteplicità** dalla parte del "tutto"

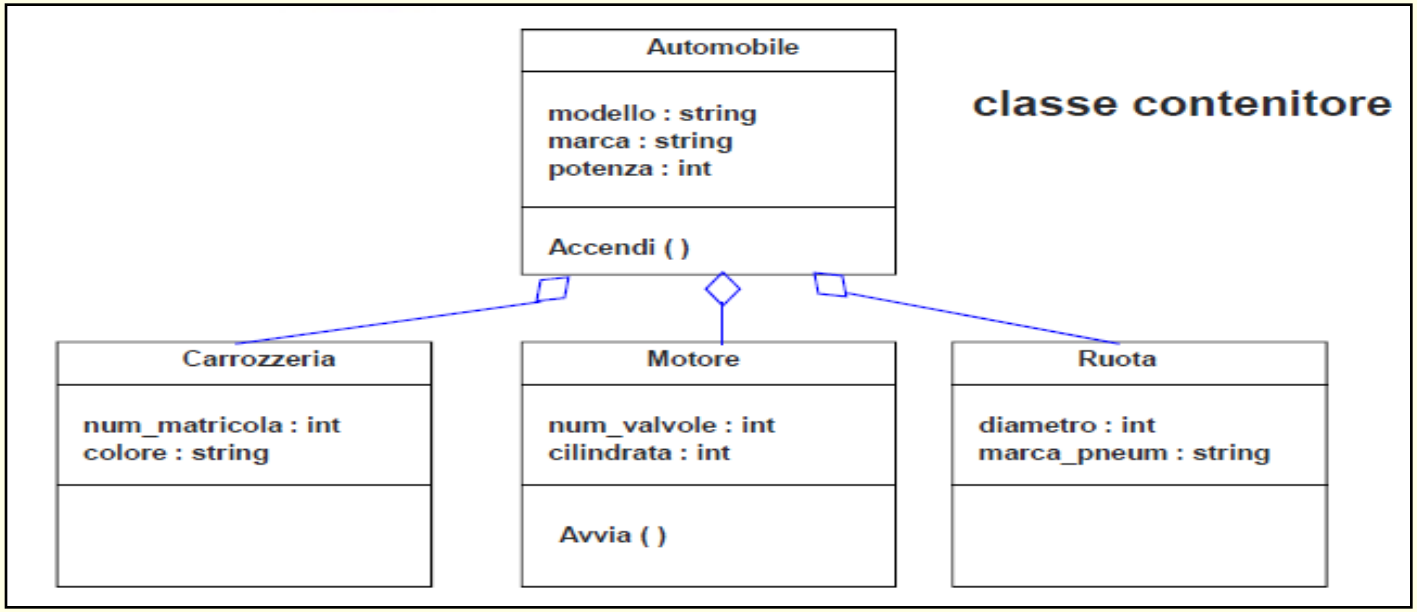


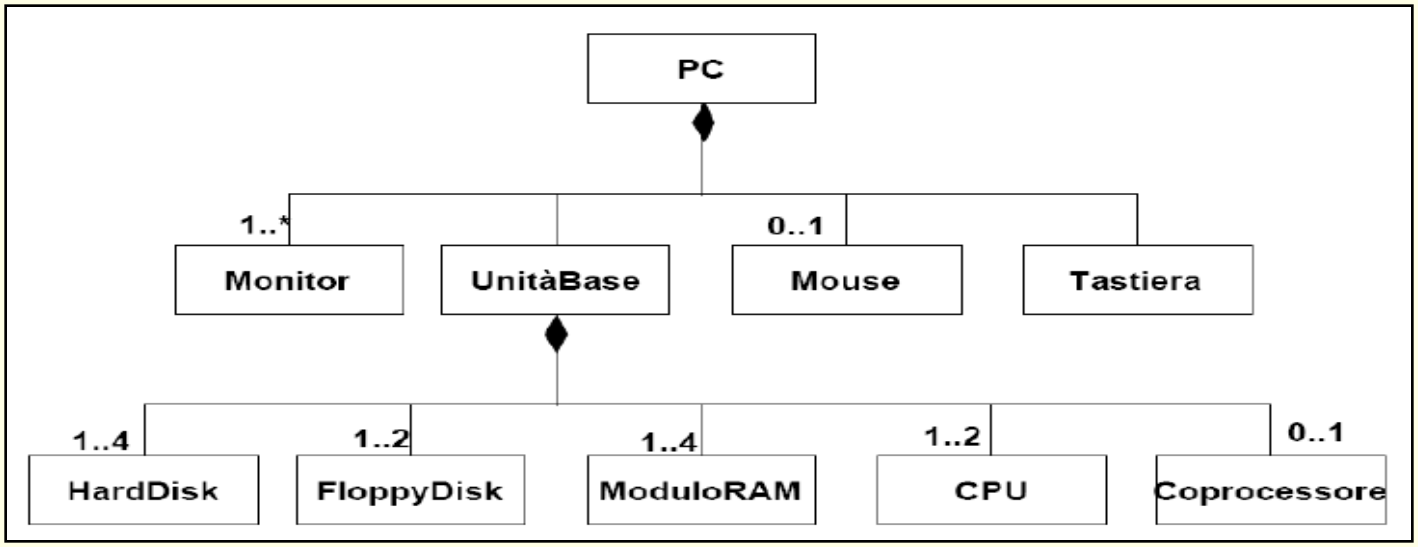
- Il **tempo di vita** (life-time) della "parte" NON è subordinato a quello del "tutto" in quanto le sue "parti" esistono indipendentemente; quando l'oggetto ottenuto aggregando altri oggetti viene distrutto gli oggetti che lo caratterizzano rimangono comunque in vita.
  - L'oggetto "**tutto**" non è responsabile della **creazione** e della **distruzione delle sue parti**



Aggregazione e Composizione

Esempi



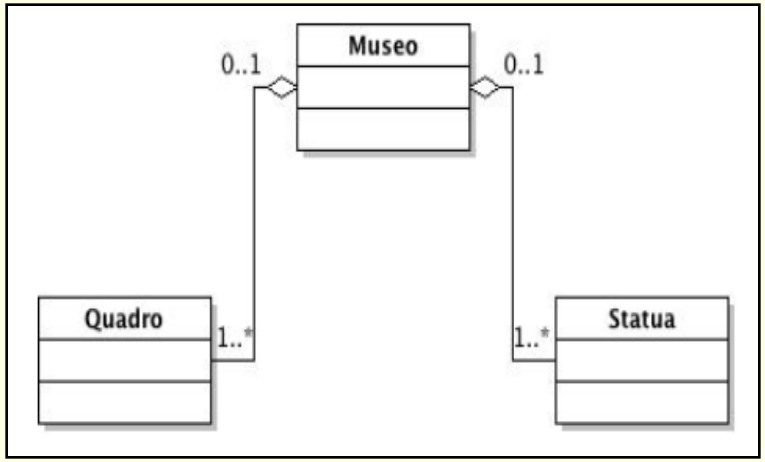


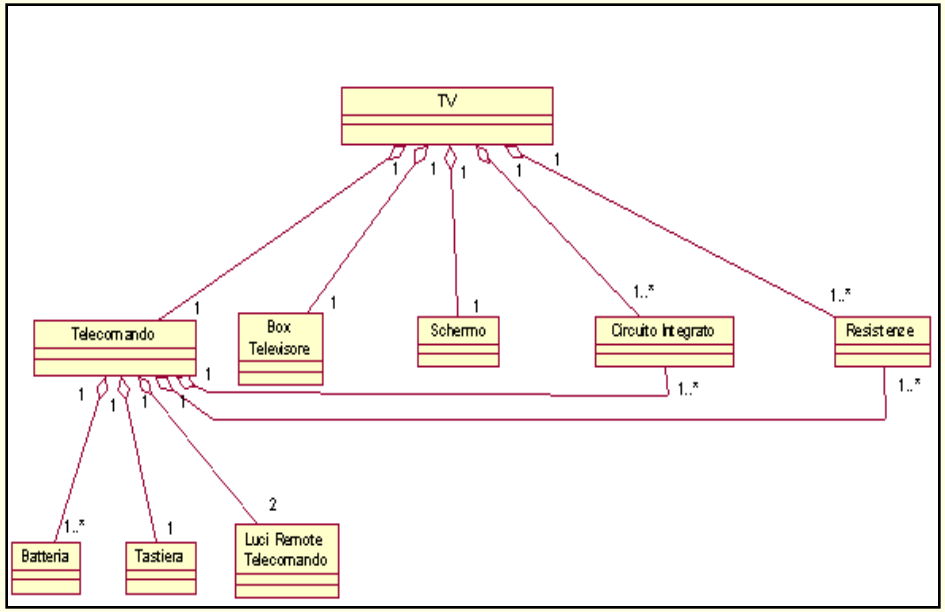
Aggregazione e Composizione

Esempi

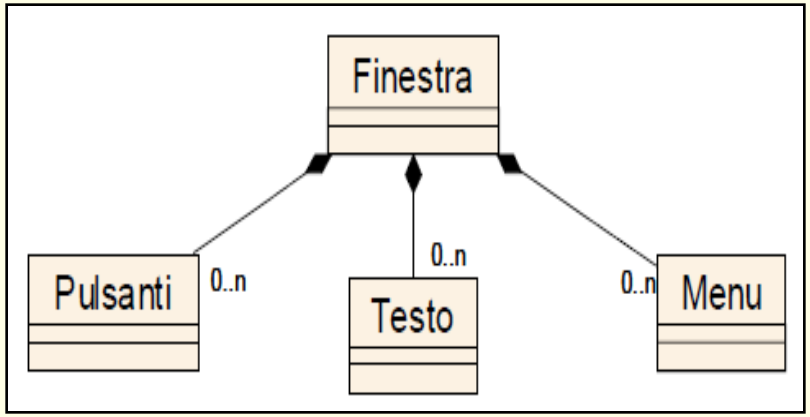
Rappresentazione con il diagramma delle classi (UML)

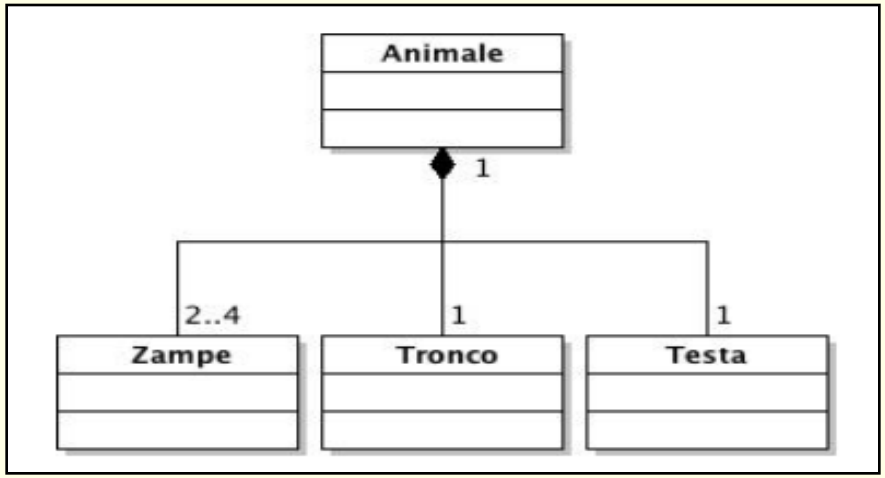
Aggregazione



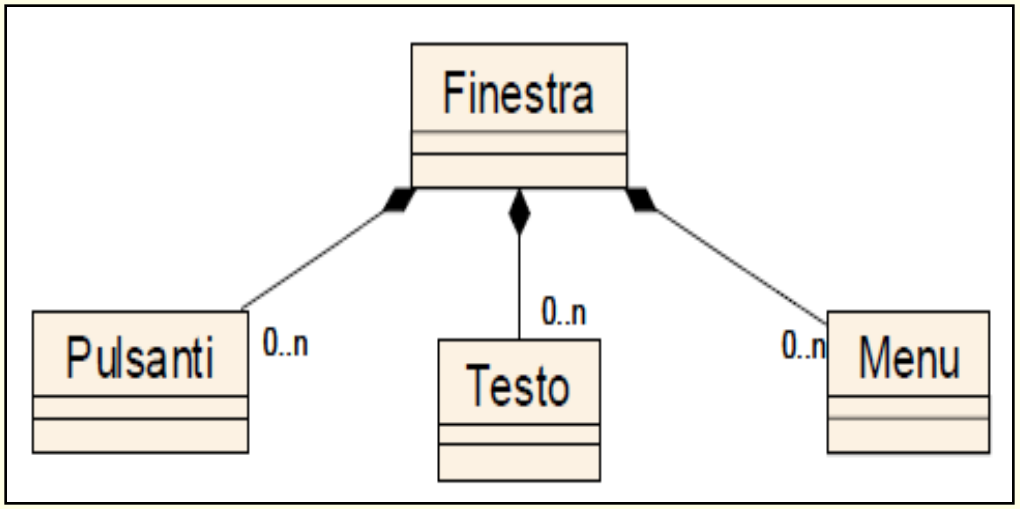


Composizione





Composizione



Implementazione:
In questo caso è come avere una classe all'interno di un'altra classe.

Per tradurre tale relazione si deve:

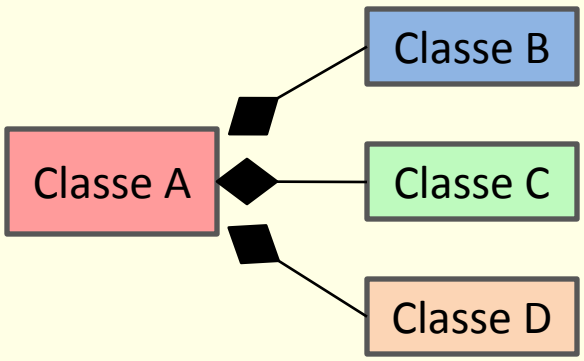
- **aggiungere una variabile membro alla classe composta** del tipo della classe componente;
- implementare il **costruttore della classe composta** in modo da richiamare il costruttore della classe componente
- il distruttore, o un suo sostitutivo, della classe composta provvederà a distruggere anche gli oggetti componenti

Composizione

Esempio

La **classe A** è formata da istanze delle classi B, C, D (precedentemente definite) messe in relazione per **composizione.**
Per rispettare questa relazione il programmatore non può mai creare istanze di B, C e D al di fuori di un'istanza di A; quindi deve creare nella classe A le istanze di B, C e D che costituiscono gli attributi dell'oggetto A.

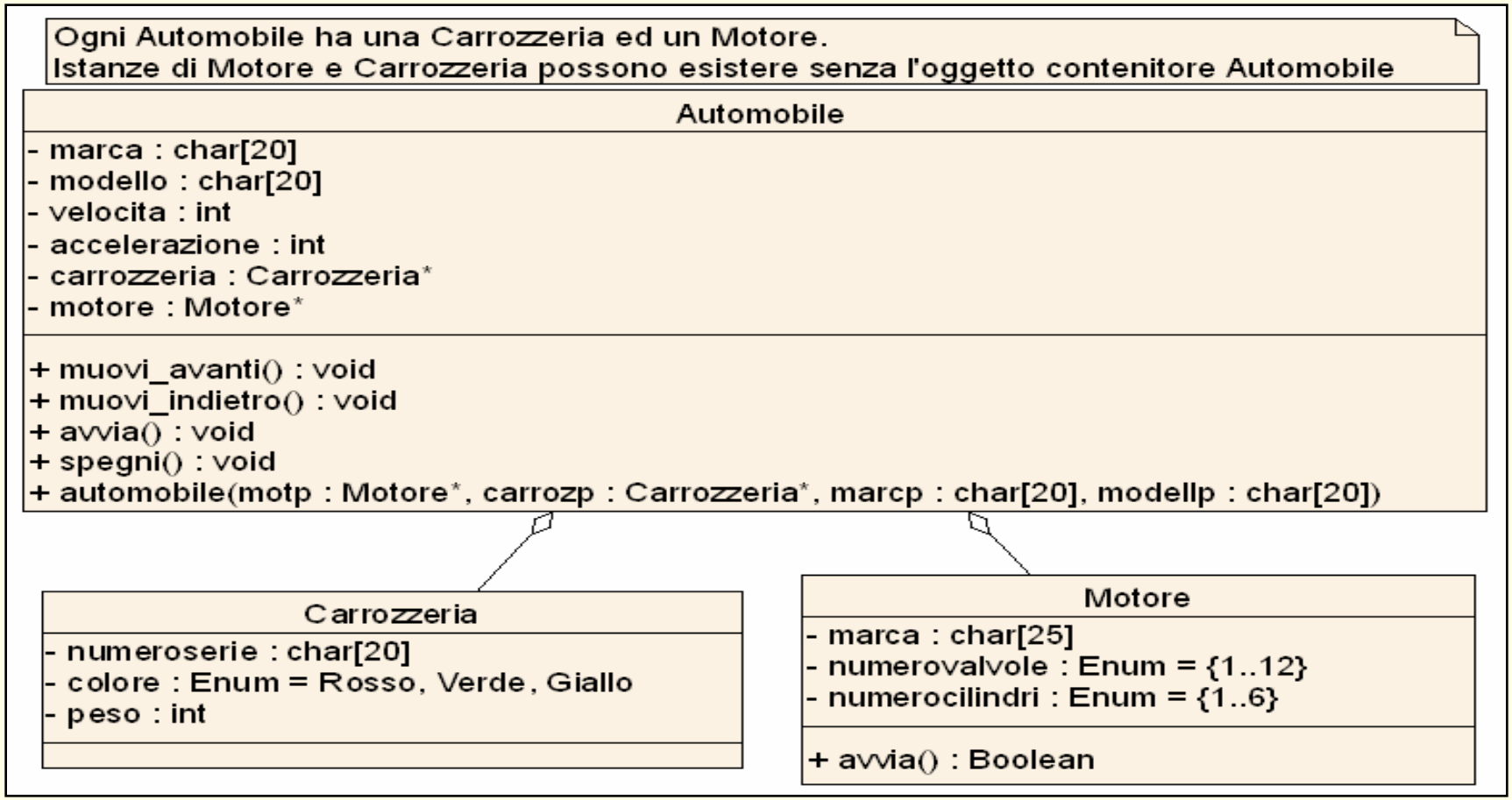
```
class A {
   private:
        B  oggettoB;
        C  oggettoC;
        D  oggettoD;
   public:
        void creaB(...);
        void creaC(...);
        void creaD(...);
 };
```




```
int main( ) {
 A  a;

 a.creaB(...);
 a.creaC(...);
 a.creaD(...);
}
```

Aggregazione



Implementazione:
Per realizzare l'aggregazione è necessario:

- **definire dei riferimenti alle istanze aggregate** (puntatori) nella parte privata dalla classe aggregante (il numero di riferimenti dipende dalla molteplicità)
- definire delle operazioni per esportare i riferimenti delle istanze aggregate

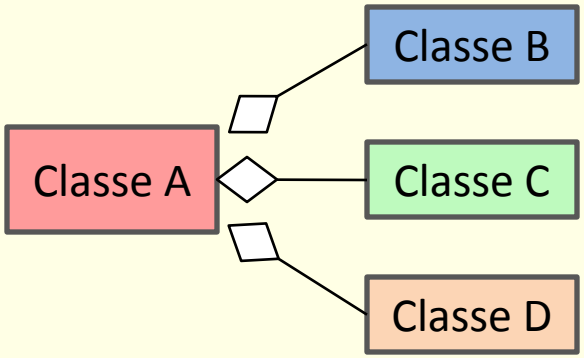
Aggregazione

Esempio

La **classe A** è formata da istanze delle classi B, C, D (precedentemente definite) messe in relazione per **aggregazione**.

Il programmatore per utilizzare la classe A rispettando la relazione di aggregazione nel main crea prima un'istanza di A e le istanze di B, C e D e poi li associa ai componenti dell'oggetto con i metodi **set**:

```
class A {
   private:
        B*  oggettoB;
        C*  oggettoC;
        D*  oggettoD;
   public:
        void setB(B*  oggB);
        void setC(C*  oggC);
        void setD(D*  oggD);
 };
```



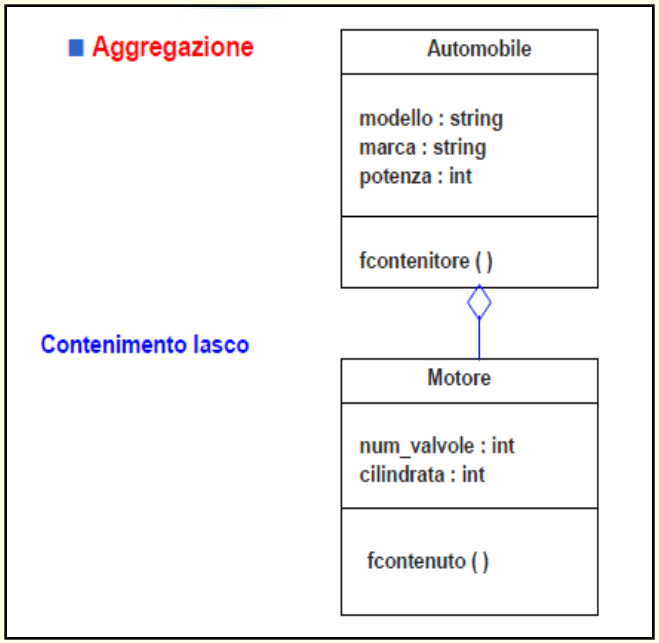

```
int main( ) {
 A  a;

 B*  b = new B();
 C*  c = new C();
 D*  d = new D();

 a.setB(b);
 a.setC(c);
 a.setD(d);
}
```

Aggregazione

Esempio



Contenitore

```
#include "Motore.h"    /* serve per inserire all'interno della classe la
                           classe Motore*/
class Automobile {
private:
    Motore* c;                  // puntatore alla classe Motore
public:
    Automobile(Motore* q):c(q){ };
    void fcontenitore( ) {
          c -> fcontenuto( );
    };
};
```

- Automobile(Motore* q):c(q){ }; è il costruttore della classe Automobile il quale usa il costruttore della classe Motore a mezzo del puntatore precedentemente dichiarato→viene inizializzato il puntatore c all'indirizzo dell'oggetto puntato da q appartenente alla classe Motore
- Automobile::fcontenitore() è una funzione appartenente alla classe Automobile
- c -> fcontenuto(); permette di utilizzare la funzione membro fcontenuto() di Motore all'interno della classe Automobile

Contenuto

```
class Motore {
public:
   Motore( );
   void fcontenuto( );
private: // variabili membro
 };
```

- Motore::Motore( ) è il costruttore della classe
- Motore::fcontenuto( ) è una funzione pubblica della classe Motore
- nella parte privata vanno inserite le variabili membro della classe

Una volta effettuato ciò l'utente della classe Contenitore deve:

1. creare l'oggetto contenuto
2. definire e inizializzare un puntatore ad esso
3. costruire l'oggetto contenitore passandogli il puntatore al contenuto

```
int main( ) {
    Motore c(lista valori iniziali);
    Motore* ptr=&c;
    Automobile A(ptr);
    A.fcontenitore(); /* tale chiamata opera su un oggetto contenitore e
                         indirettamente sul contenuto */
}
```

Aggregazione e Composizione