Programmazione orientata agli oggetti

- Origini del modello a oggetti:
  - Linguaggi di Programmazione
  - Sistemi Operativi
  - Intelligenza Artificiale

- Esigenza di una metodologia di strutturazione dei programmi basata sulla ***protezione*** e sul ***riutilizzo delle informazioni***

- Tutto cio` ha portato alla definizione del concetto di ***oggetto***

Oggetto

- Collezione di ***operazioni*** che condividono uno ***stato*** [Wegner, 90]

```
point:object
   x:=0; y:=0;
   read_x: -> x;     {restituisce il
                      valore di x}
   read_y: -> y;     {restituisce il
                      valore di y}
   change_x(dx): x:=x+dx;
   change_x(dy): y:=y+dy;
```

- Linguaggi di programmazione ad oggetti, veri e propri linguaggi "general-purpose"

Aspetti fondamentali:

- Distinzione tra ***classe*** ed ***istanza*** (anche metaclassi);
- ***Descrizioni*** di classi in termini di proprieta` (default) che ammettono ***eccezioni***
- Supportano ***conoscenza procedurale*** come ***metodi*** associati agli oggetti

Tecnologia ad Oggetti:

anni 60 - ***inizi*** (Simula67)

Aggancio con il filone della

Programmazione modulare
Tipo di dato Astratto

anni 70 - *Goldberg* (Smalltalk)
unico ambiente integrato e grafico
orientato allo ***sviluppo rapido***
di applicazioni

anni 80 - *Meyer* (Eiffel)
linguaggio ad oggetti con accento
sulla ***correttezza*** ed il ***controllo statico***

Stroustrup (C++)
estensione del linguaggio C con classi
proposta ibrida con implicazioni e
diffusione molto significative
orientato allo ***sviluppo di sistema***

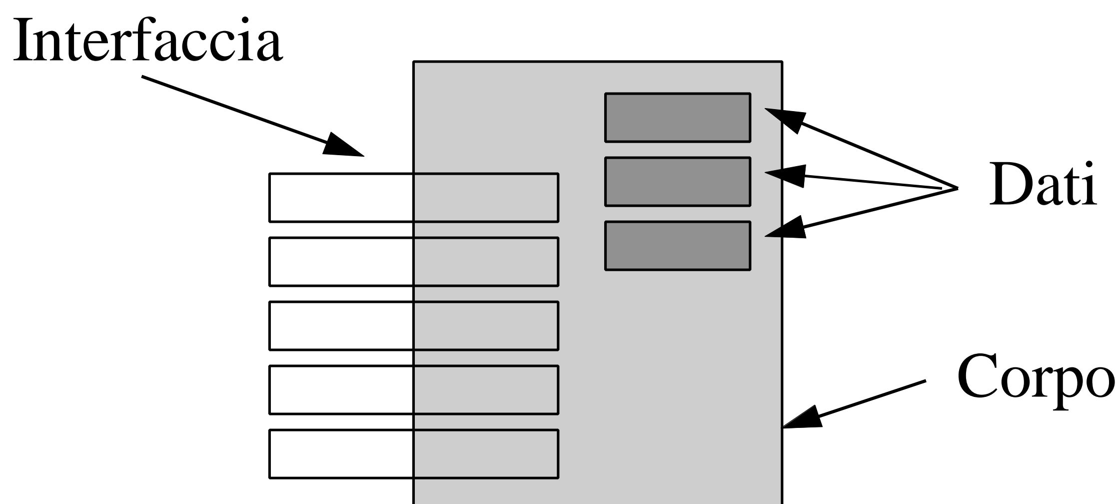
Caratteristiche principali:

- Modularita`
- Protezione (information hiding)
- Riusabilita`
  - Ereditarieta`
  - Polimorfismo
  - Genericita`
- Dinamicita`

Caratteristiche principali (cont.):

- **Riusabilita`**
  - La possibilita` di riutilizzare un intero sistema software o parti di esso nello sviluppo di altri prodotti software

- **Modularita` e Protezione**
  - Un modulo consiste di un *interfaccia* e di una *implementazione*



  - Chi usa il modulo conosce solo l'interfaccia
  - L'implementazione e` *nascosta*

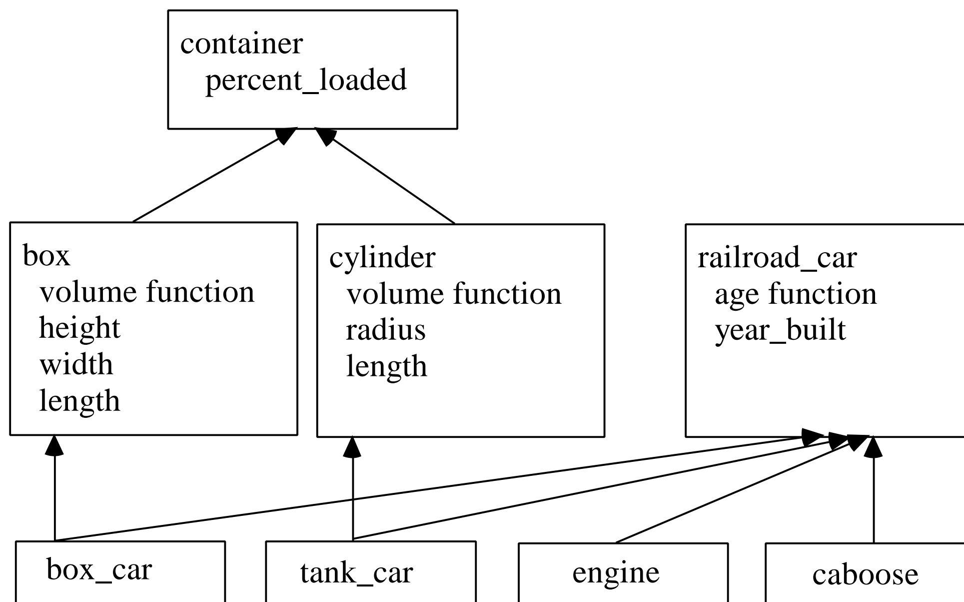
• Modularita` (cont.)

- Tre categorie di moduli
  - Moduli che offrono solo funzioni e procedure e non contengono dati (librerie)
  - Strutture Dati Astratte (ADS, astrazioni di dato): una ADS consiste di dati e funzioni: i dati sono *nascosti* e possono essere modificatii solo dalle funzioni del modulo
  - Tipi di Dati Astratti (ADT): un ADT e` simile a una ADS, ma rappresenta un tipo di dato

- **Tipo di Dato Astratto**
  - Un ADT definisce una struttura dati e l'insieme delle operazioni che possono essere eseguite su una particolare istanza dell'ADT
  - I dati e l'implementazione sono *privati* in modo da non poter essere alterati accidentalmente
  - Un ADT funziona esattamente come un tipo di dato di base, quale **int** o **float**

Per evitare la duplicazione di informazioni, alcuni linguaggi (Simula67) introducono una relazione di ***ereditarieta`*** di comportamenti fra tipi di dati astratti (***classi***)

- **Ereditarieta`**
  - Meccanismo per costruire nuovi ADT partendo da quelli esistenti *ereditandone* le proprieta`
  - Riutilizzo del codice



  - Una classe C2 puo` essere ottenuta come sottoclasse di una classe C1, dichiarata precedentemente, riutilizzando cosi` le definizioni di informazioni ed operazioni di C1
  - Un'istanza di C2 avra` non solo i comportamenti (variabili ed operazioni) specificati localmente in C2, ma anche quelli ereditati dalla classe progenitrice C1, e cosi' via percorrendo la catena delle superclassi

- **Ereditarieta` (cont.)**

  - Nei primi linguaggi con classi (Simula67, Alphard) la relazione di ereditarieta` e` ***a singolo genitore***, cioe' una classe puo` ereditare al piu` da una ed una sola superclasse

  - Gerarchia di ereditarieta`: ***albero***

  - Programmazione per differenza

  - Le sottoclassi ***aggiungono*** codice o informazioni o ***ridefiniscono*** comportamenti o significati dei dati

- **Polimorfismo**

  - Dal greco "pluralita` di forme"

  - Permette di utilizzare lo stesso *nome* per identificare operazioni simili che differiscono per la realizzazione

  - Una sola *interfaccia* per molte azioni (*metodi*)

    Ad esempio, **`push`** e **`pop`** per stack di **`int`**, **`float`**, **`char`** (*overloading*)

  - Il compito di selezionare l'azione specifica da applicare viene delegato al compilatore

- **Genericita`**

  - Possibilita' di definire moduli o classi parametrici

  - Una classe (modulo) generica e` definita in termini di alcuni "parametri formali"

  - La classe (modulo) viene istanziata solo quando si specificano i parametri attuali corrispondenti ai formali

    Ad esempio, stack di elementi di tipo T:

    ```
    class stack (T)
    ```

    da cui si istanziano:

    ```
    stack (int)     SI;

    stack (float)SR;
    ```

- ***Overloading,*** incarica il sistema di ritrovare la realizzazione corretta

vs.

- ***Genericita`***, si usa un modulo generico attraverso le sue istanze, che non sono piu` flessibili.

- **Dinamicita`**

  - Creazione e distruzione di oggetti a tempo di esecuzione

  - Valutazione a tempo di esecuzione dell'identita' degli oggetti (*dynamic binding* e *polimorfismo orizzontale*)

  - Variazioni comportamento (*polimorfismo verticale*)

Collegamento tra entita`:

- In un ambiente di programmazione in-the-large, il riferimento fra entita` distinte puo' essere:

  - risolto prima dell'esecuzione (*binding statico*) (ad es., MODULA2)

  - determinato solo all'esecuzione (*binding dinamico*) (ad es., SMALLTALK e metodi virtuali in C++)

Programmazione a Oggetti

- La Programmazione a Oggetti (Object-Oriented Programming, OOP) e` uno *stile di programmazione*

- Un linguaggio *supporta* uno stile di programmazione se fornisce gli strumenti per renderne conveniente (facile, sicuro ed efficiente) l'impiego

- Il programmatore specifica ***cosa*** fare con un oggetto piuttosto che concentrarsi su ***come*** qualcosa viene fatto

Terminologia

- **Classe**

  - costrutto base per definire tipi di dati astratti

  - *member variables*: descrivono i dati contenuti nel tipo di dato astratto (STATO)

  - *member functions*: definiscono le OPERAZIONI possibili sul tipo di dato

  - Solo alcune delle variabili ed alcune delle operazioni verranno rese visibili all'esterno (*interfaccia*)

  - Invisibilità della realizzazione del servizio

  - Separazione tra:

    interfaccia e
    realizzazione

• Classe (cont.)

- "Template" da cui possono essere creati gli oggetti (***tipi***)

- Per raggruppare e riusare ***comportamenti comuni*** di alcune risorse in un unico contenitore

- Una classe descrive un insieme di oggetti che
  - hanno la stessa ***interfaccia*** ed
  - hanno gli stessi ***attributi*** interni

- La classe **`point`** ha le stesse variabili di un oggetto istanza, ma con diversa interpretazione (variabili "potenziali")

```
point: class
    {variabili locali delle istanze}
    {operazioni o metodi}
```

- Copia delle variabili locali per ciascuna istanza, mentre il codice dei metodi e` condiviso

Terminologia (cont.)

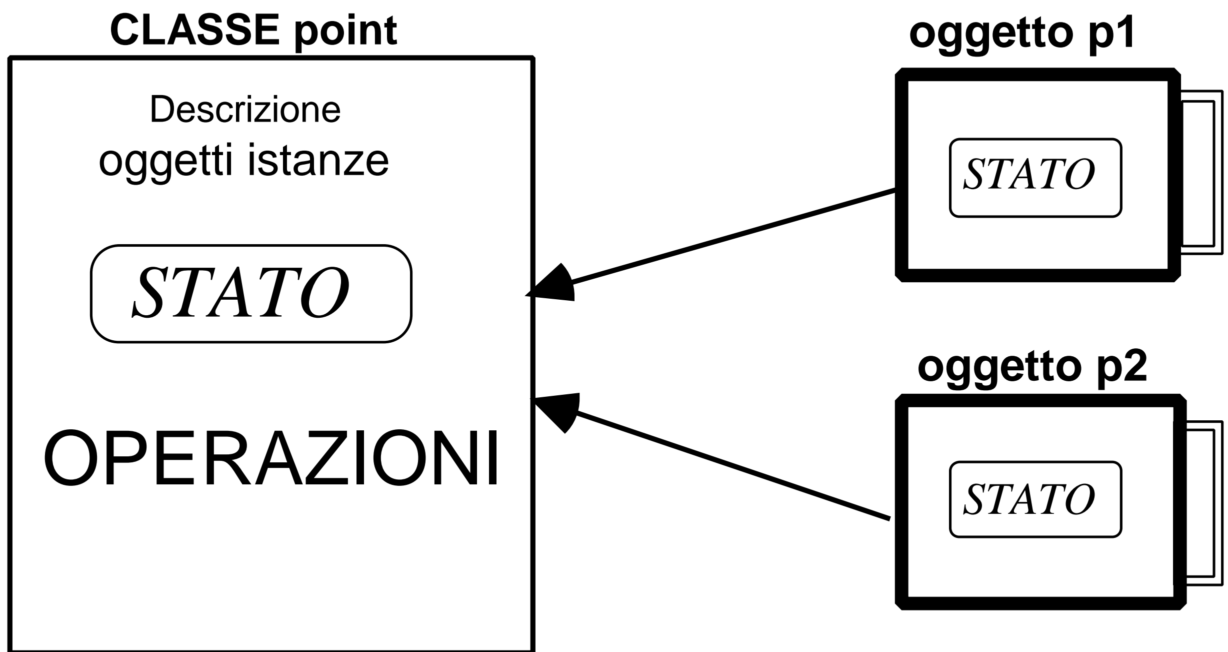
• Oggetto

Entita` base della programmazione Object-Oriented

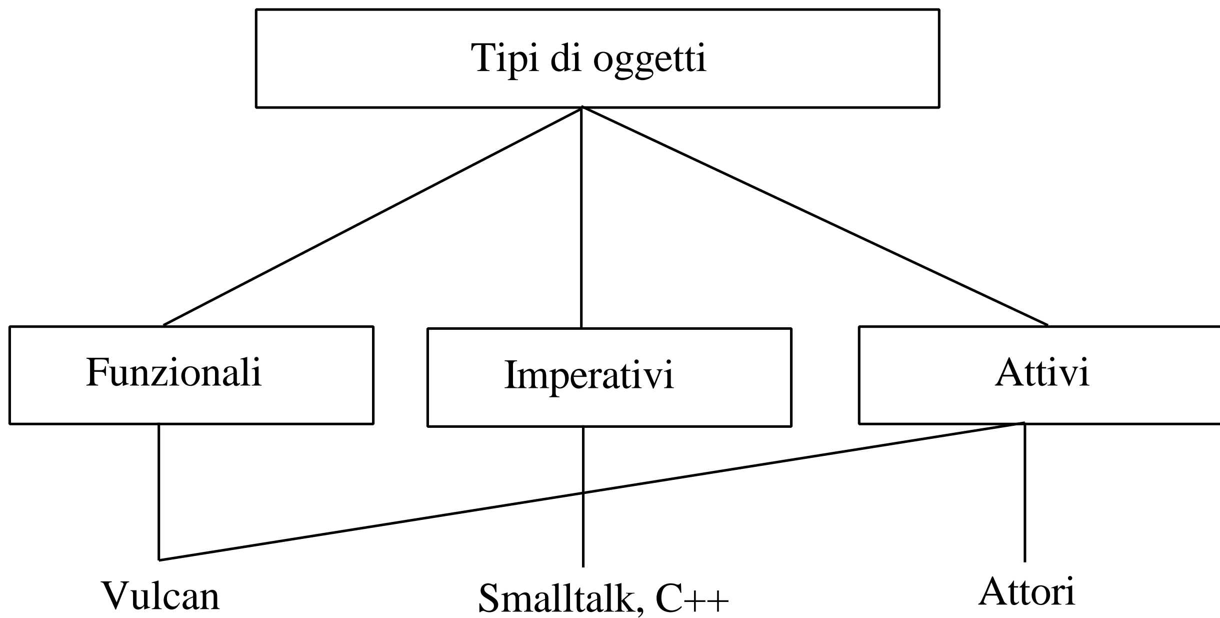
- istanza di un ADT che possiede ***proprieta`*** (*variabili di istanza*) e fornisce operazioni (***metodi***) definite per quel particolare tipo di dato

- L'operazione di creazione di un'istanza crea una copia delle variabili

```
p1:=make_instance_point(0,0);
p2:=make_instance_point(1,1);
```


CLASSE point
Descrizione
oggetti istanze
STATO
OPERAZIONI
oggetto p1
STATO
oggetto p2
STATO

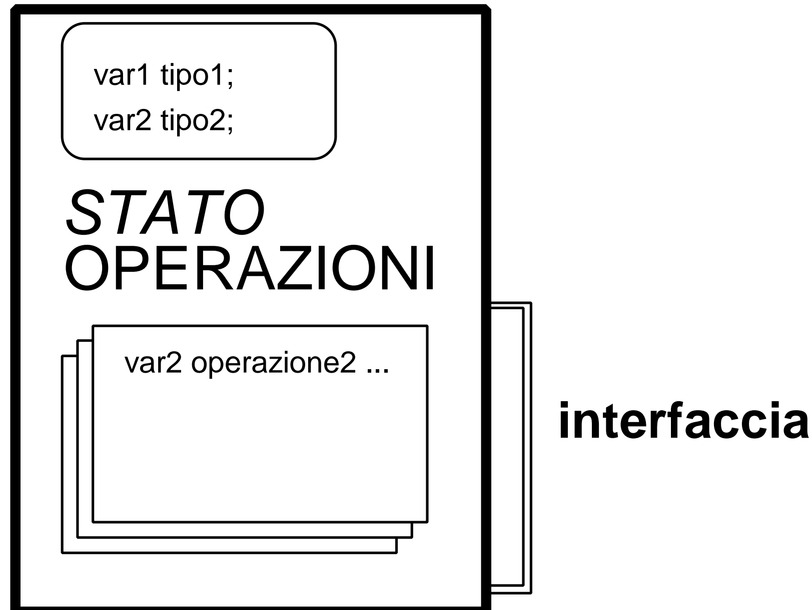
Tipi di oggetti: [Wegner, 1990]



- Funzionali, non hanno stato modificabile
- Imperativi, stato aggiornabile attraverso operazioni
- Attivi, gli oggetti sono processi attivi alla ricezione del messaggio (necessita` di sincronizzazione)

Variabili:

Possono essere con tipo o meno



ambienti per il rapido sviluppo prototipale

Smalltalk
nessun tipo associato alle variabili
controlli di correttezza ***dinamici***

ambienti con maggiore controllo

Eiffel, C++
tipo associato alle variabili
controlli di correttezza ***statici*** e ***dinamici***

- ***Semantica per riferimento***

  Una variabile non puo` contenere un oggetto, ma solo un ***riferimento*** ad esso

  In un modello ad oggetti "puro", la creazione di un oggetto restituisce il puntatore all'oggetto creato (Smalltalk, Java)

  Un oggetto contiene nel proprio stato solo riferimenti ad altri oggetti o valori primitivi (tipi predefiniti)

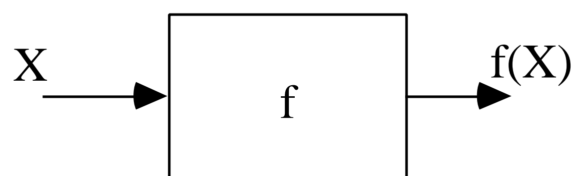
- ***Semantica per valore***

  Una variabile contiene l'intera "struttura" di un oggetto (e` il default in C++)

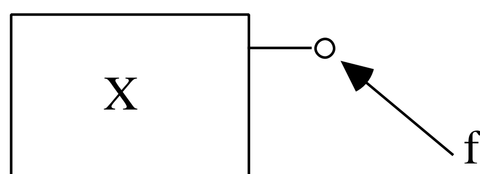
  Per ottenere quella per riferimento, uso esplicito di ***puntatori*** in C++

- **Messaggio**
  - un oggetto esegue determinate azioni (***metodi***) in risposta a un ***messaggio***
  - l'azione eseguita dipende dal ***messaggio*** e dall'***oggetto*** che riceve il messaggio

- **f(X)** si applica una funzione f all'argomento X



- **X.f** spedisce un messaggio all'oggetto X per richiedere l'operazione f



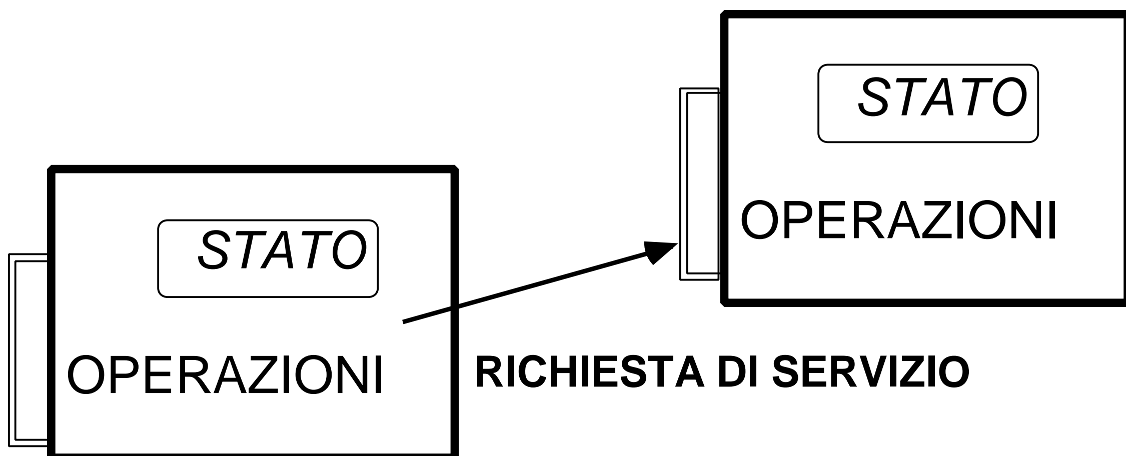
Calcolo orientato ai dati

Operazioni su un oggetto

- richiesta di eseguire un metodo (messaggio)
- richiesta di valore di un attributo
- variazione di valore di un attributo

Servizio

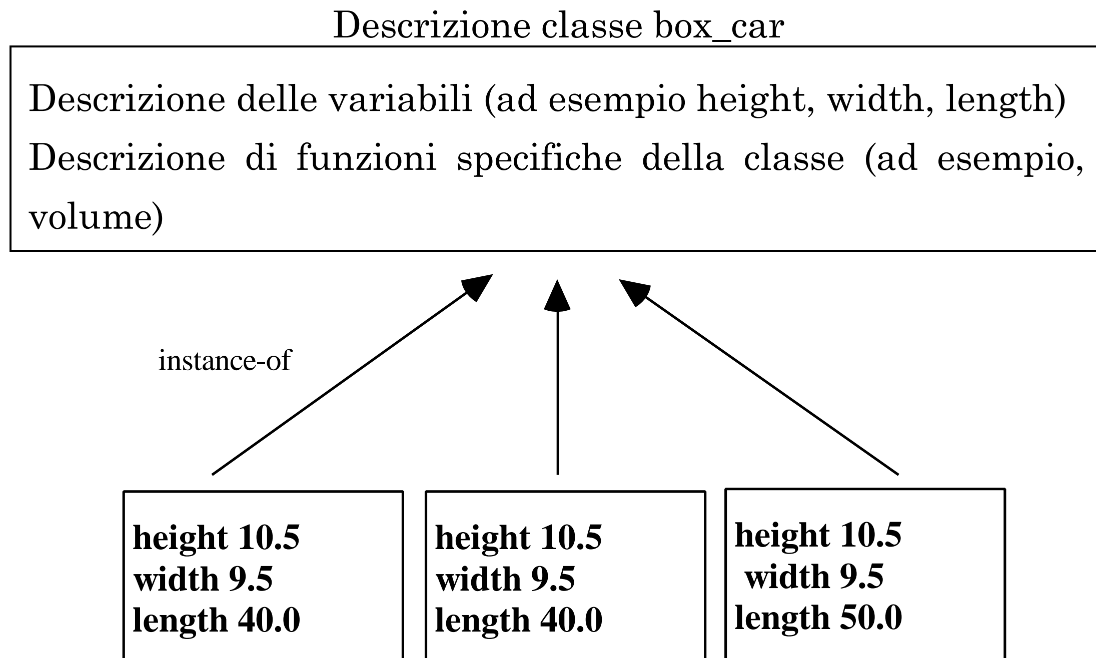
- Modello cliente/servitore
- L'identificazione del servitore è fatta al momento della richiesta (***dinamicità***)



Identificazione oggetti

- Necessita` di un nome identificativo ***unico*** nell'***ambiente di esecuzione***
- Il ***nome del servitore*** è accessibile attraverso le ***variabili*** dell'oggetto che effettua la richiesta
- Generalmente non e` possibile identificare un oggetto in base al suo comportamento o variabili
- Esistono pero` anche ***sistemi di nomi*** molto sofisticati ed evoluti

Esempi C++:




```
class box_car {
public:
    double height, width, length;
};
```

- ***Member variables***, variabili (istanza) che compaiono nella definizione della classe

E` possibile specificare valori di default

Dichiarazione di un'istanza:

```
    box_car X;
```

Accesso ai valori di una variabile (di istanza):

```
    X.height=10.5;
```

Creazione dinamica mediante puntatori

```
box_car *P;
P = new box_car;
```

```
#include <iostream.h>

class box_car {
public:
   double height, width, length;
   double volume()
          {return height*width*length;}
};

class tank_car {
public:
   double radius, length;
   double volume()
      {return pi*radius*radius*length;}
};

main()
{ box_car X;
  X.height=10.5;
  X.width=9.5;
  X.length=40.0;
  tank_car Y;
  Y.radius=3.5;
  Y.length=40.0;
  cout << "The volume of box_car is"
       << X.volume() << endl
       << "The volume of tank_car is"
       << Y.volume() << endl;
}
```

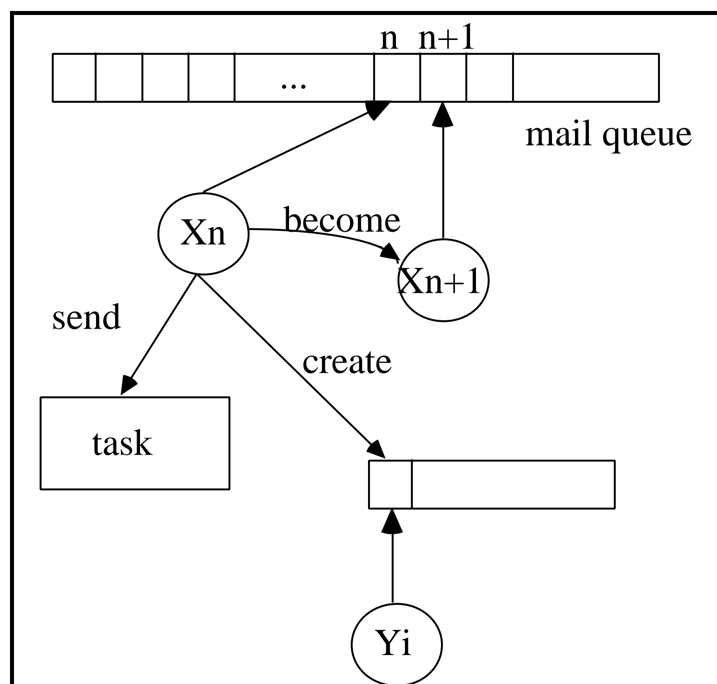
- Nessun parametro di un metodo puo` riferire l'oggetto o una delle sue variabili

- Ci possono essere argomenti "ordinari"

Attori

- Nel modello ad attori non esistono classi, ma solo istanze (***prototipi***)

- Un prototipo e` un oggetto dotato di un ***comportamento*** ed uno ***stato*** che puo` essere utilizzato per creare altre istanze



- Un attore ha un ***indirizzo***, una ***mailbox*** che mantiene una coda dei messaggi ed un ***comportamento***

- Ad ogni ciclo, letto il prossimo messaggio, si puo` eseguire una delle seguenti azioni:
  - creazione di nuovi attori (**`create`**)
  - spedizione di messaggi ad altri attori (**`send`**)

- modifica del comportamento (**`become`**)

Gerarchie di classi e oggetti

- Gli oggetti vengono organizzati in una ***gerarchia*** o ***ordinamento parziale*** detta ***gerarchia di ereditarieta`*** o ***tassonomia***

- ***Oggetti*** e ***classi*** rappresentati da nodi

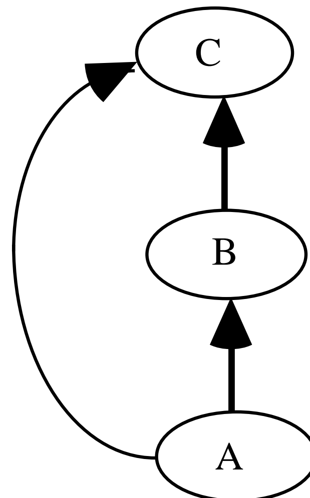
- ***Relazioni gerarchiche*** tra oggetti rappresentate connettendo nodi via archi "is-a" o "instance-of"

- Cattura il principio base di organizzazione di molti sistemi (anche di Intelligenza Artificiale, quali reti semantiche, sistemi a frames), fra cui sistemi ad oggetti

- La ***tassonomia*** viene utilizzata per memorizzare informazioni al livello piu` appropriato di generalita`, rendendole automaticamente disponibili agli oggetti piu` specifici mediante il meccanismo di ***ereditarieta`***

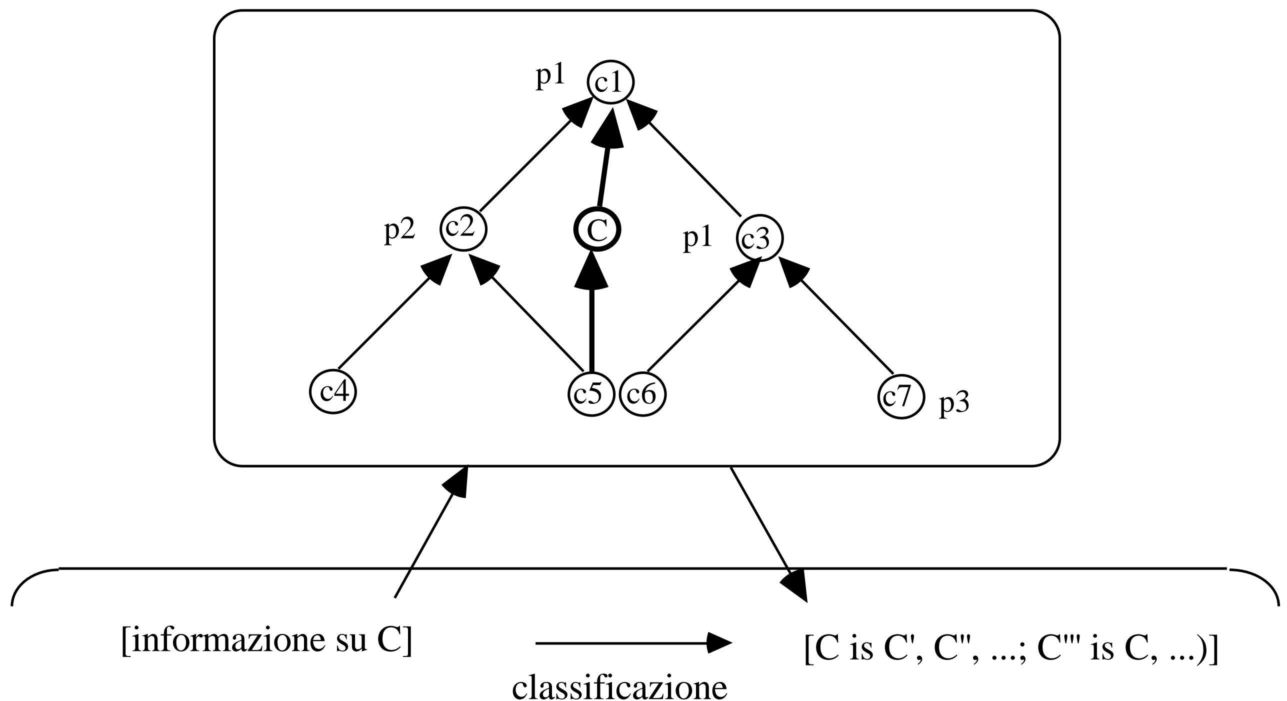
- La ***tassonomia*** puo` essere ***costruita***:
  - ***direttamente*** dal programmatore, asserendo i legami "is-a" tra coppie di concetti (reti semantiche, sistemi a frames e ad oggetti)
  - ***automaticamente*** inferendo i legami "is-a" tra coppie di concetti (ad esempio, nei linguaggi terminologici)

- Costruita la tassonomia, appartenenza ad una classe per chiusura transitiva della relazione "is-a" (is-a*)



Classificazione:

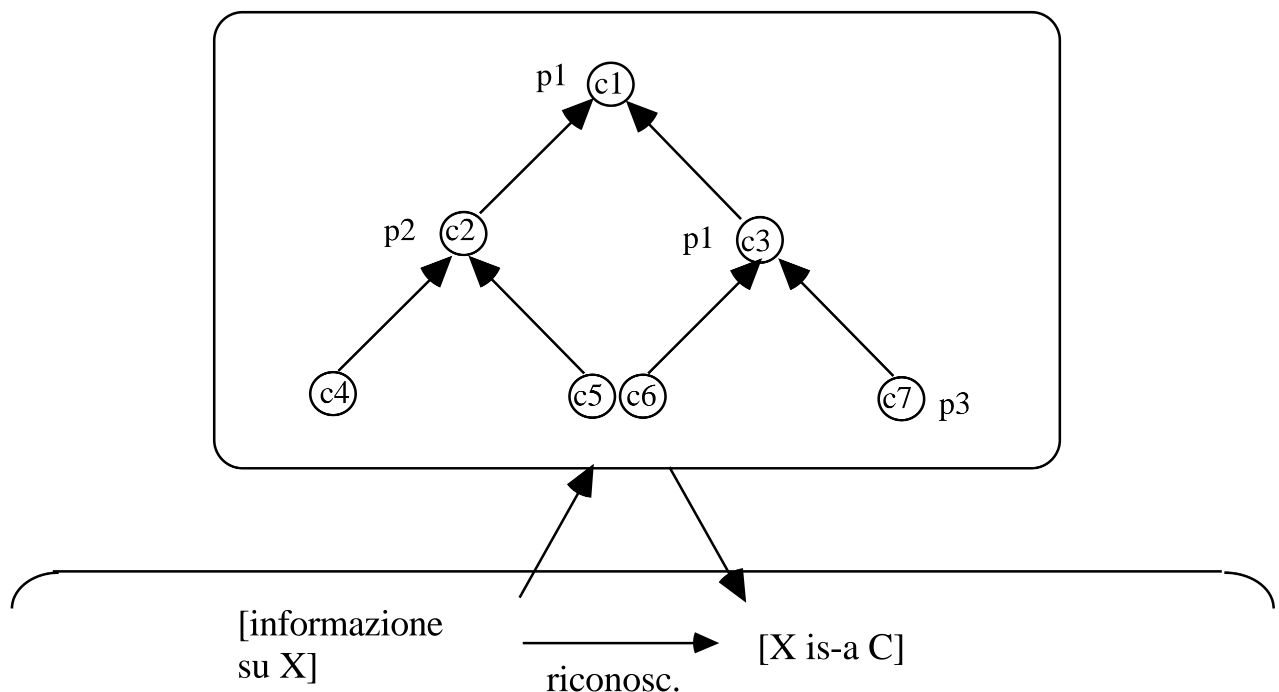
- Inserire una classe nella tassonomia al livello piu` appropriato nella gerarchia



- Compito delegato al programmatore nel caso di sistemi OO

Riconoscimento:

- Data una tassonomia, e` l'operazione con cui si "classificano" individui cioe` istanze (oggetti)



- Nel caso di linguaggi OO, creazione dinamica di istanze

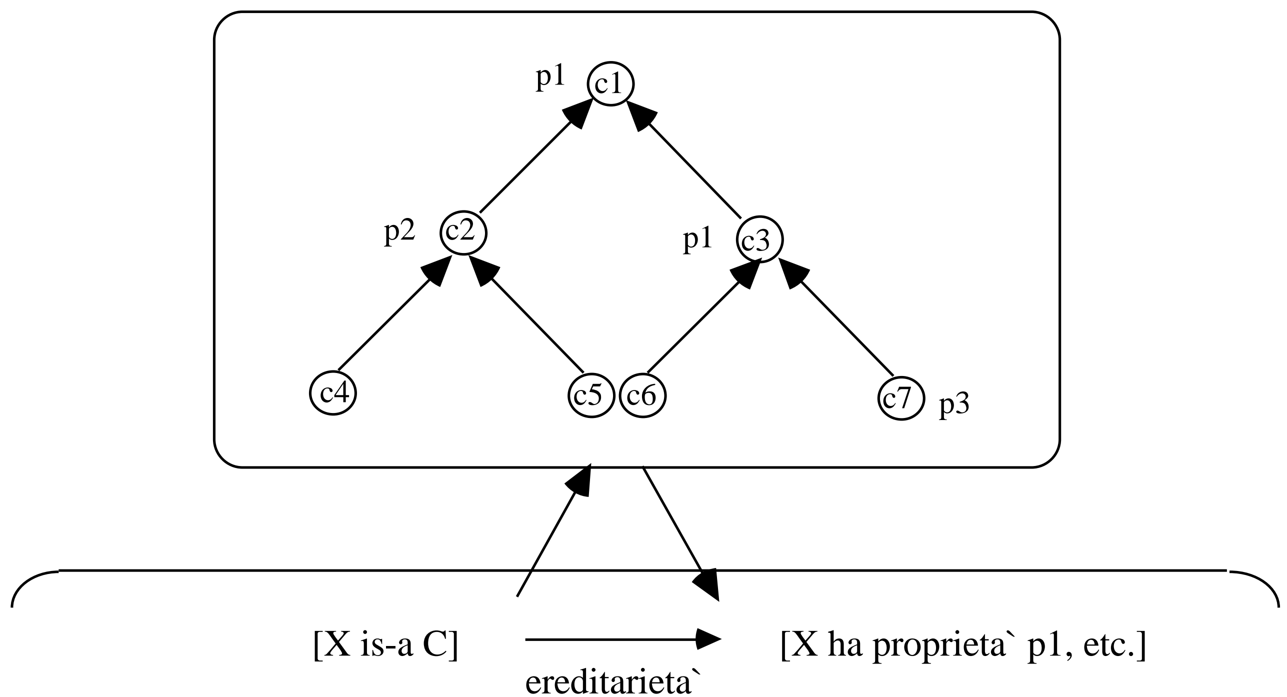
```
P = new box_car;
```

- Determinazione a tempo di esecuzione del codice da applicare in risposta ad un messaggio inviato a **`P`**

Ereditarieta`

- E` una forma di inferenza che consente di determinare le proprieta` di un oggetto basandosi sulle proprieta` dei suoi ***antenati***

  Gli uccelli volano
  Titti e` un uccello
  Titti vola



- Presenza di proprieta` in conflitto (eccezioni, ereditarieta` multipla)

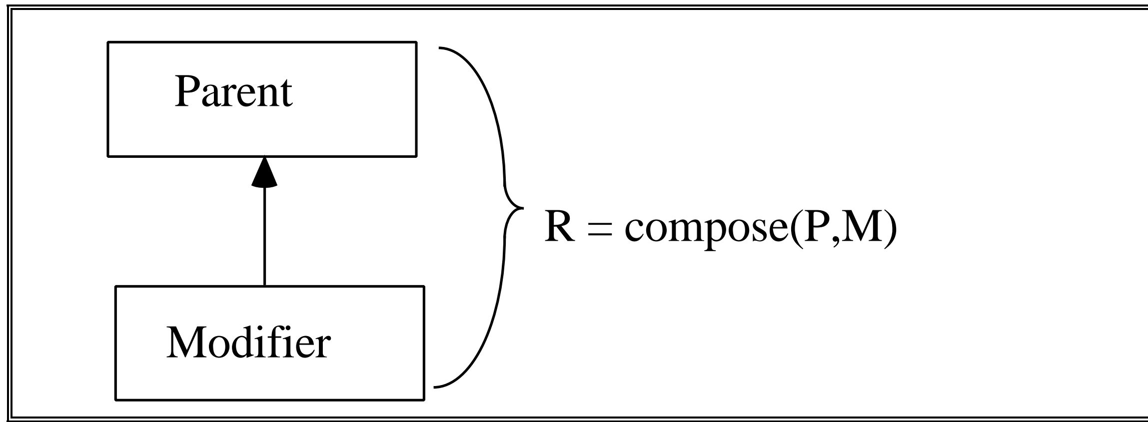
Ereditarieta` (cont.)

- Nell'ambito dei ***linguaggi a oggetti***
  - l'ereditarieta` consente di riutilizzare il comportamento di una classe nella definizione di nuove classi
  - si ereditano attributi (***variabili***) e metodi (***operazioni***)
  - sia le variabili che le operazioni ereditabili possono essere ridefinite (***overriding***)
  - in alcuni casi si e` in presenza ***ereditarieta` multipla***

Due forme di ***modifica*** incrementale: [Wegner, 1990]

- raffinamento (refinement)
  B refines A — B conserva ed ***aumenta*** le proprieta` ed il comportamento di A
- somiglianza
  B like A — B ed A hanno alcune proprieta` e comportamenti comuni

- L'ereditarieta` consente di costruire un modulo software dato da un Parent piu` un Modifier:



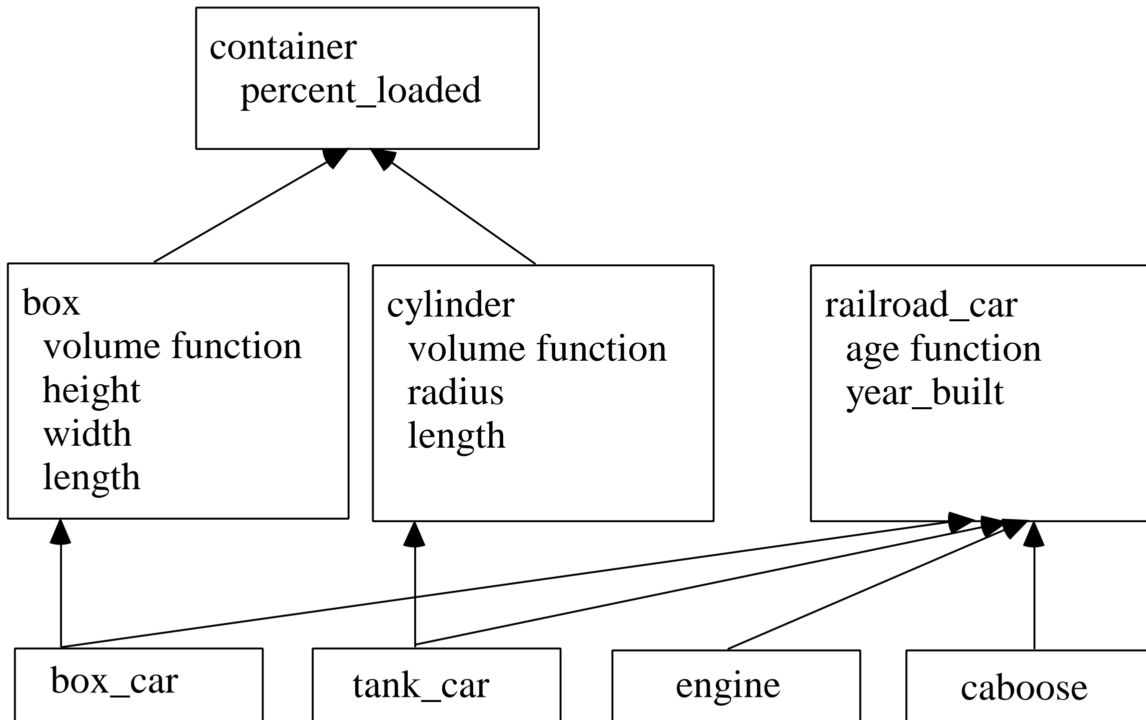
- Se gli attributi e i metodi di M sono disgiunti da quelli di P, raffinamento

- Se no, quelli di M ridefiniscono quelli di P (***overriding***)

Regole di combinazione:

- usare la prima definizione (***ovverriding statico***)
- usare tutte le definizioni
- usare il codice del primo metodo che restituisce un valore (***overriding dinamico***)
- usare un selettore per scegliere uno degli attributi o metodi delle classi genitrici (qualificazione o *scope* in C++)

Ereditarieta`: esempi C++



- Le variabili istanza (member variables) e i metodi vengono definiti al ***livello appropriato nella gerarchia***

```
class box:
   public container
   {private:
      double height, width, length;
    public:
      double volume()
         {return height*width*length;}
}

class box_car:
   public railroad_car,
   public box{public: box_car()
                  {height=10.5;
```

```
                    width=9.5;
                    length=40.0;}
};
```

Protezione:

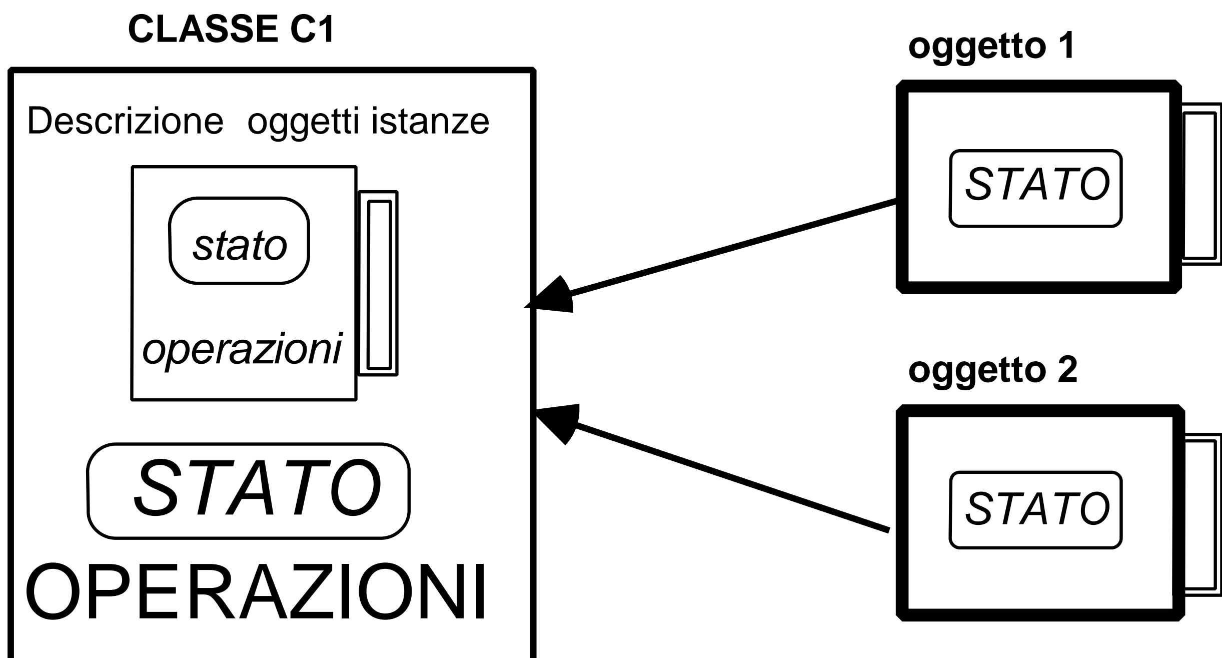
```
class tank_car {
public:
    tank_car() {radius=3.5, length=40.0};
    double read_radius() {return radius;}
    void write_radius(double r)
                              {radius=r;}
    ...
    double volume ()
        {return pi*radius*radius*length;}
private: double radius, length;
};

main()
{  tank_car X;
  X.radius=6;     /* fallimento */

  X.write_radius(6);


}
```

- Accesso alle variabili istanza ed ai metodi di una certa classe C:
  - limitato ai soli metodi di C, **`private`**
  - limitato ai soli metodi di C e sottoclassi, **`protected`**

- non limitato (pubblico a tutti), **`public`**
- accesso alle variabili in sola lettura, variabili **`private`** e metodi (reader) **`public`**

Metaclassi

- Una classe puo` essere, a sua volta, un oggetto (***metaclassi***, Smalltalk)



- Naturalmente anche le ***classi*** hanno identificatori unici se sono ***oggetti*** (Smalltalk) altrove sono ***puri descrittori*** (C++, Eiffel)

- In C++, e` possibile definire variabili che sono attributi della classe (*static variables*)

Genericita`

- Riutilizzo di codice favorito attraverso l'uso di ***classi generiche*** (*template* in C++)

- Il *template* offre un meccanismo per definire una classe ***generica***:

```
template<class Tipo>
class id_classe
{private:
    ...;        // puo` comparire Tipo
 public:
    ...;}    // puo` comparire Tipo
```

- Da classi ***astratte*** si possono definire le classi particolari ***istanziandole*** sul tipo di dato usato:

```
id_classe<id_tipo_concreto>  id_var;
```

Ereditarieta` (cont.)

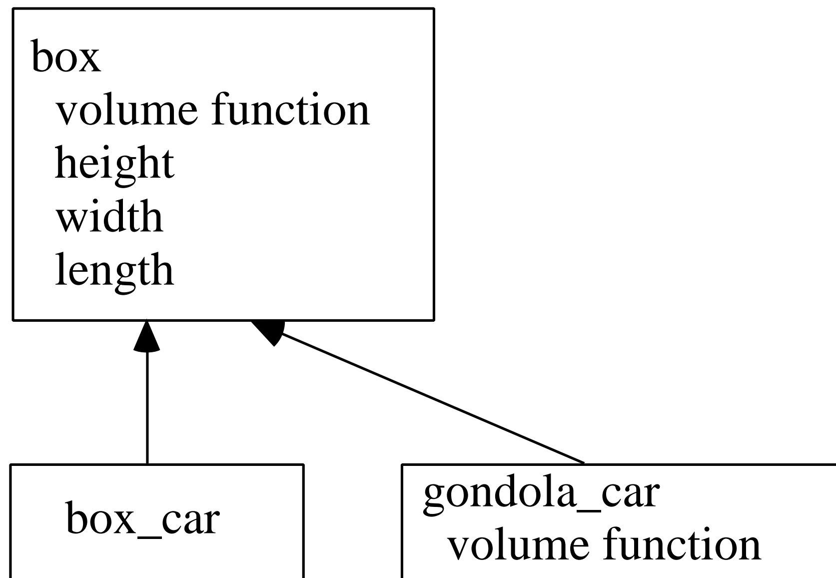
- Molti sistemi sono ***non-monotoni***
- Si ammettono ***eccezioni*** nelle sottoclassi (***overriding*** di proprieta` o di codice)

Esempio:

```
class point {
protected
    int x=0, y=0;
public:
    void change_x(int dx){x=x+dx;}
    void change_y(int dy){y=y+dy;}
    void display_point()
              {cout  << "coordinate: "
                     << x << ", "
                     << y << endl;}
}

class colored_point: public point
protected
    int color=1000;
public:
    void display_point()  //overriding
              {cout  << "coordinate: "
                     << x << ", "
                     << y << " colore "
                     << color << endl;}
```

```
}

...
colored_point P;
P.display_point();
```

Overriding:



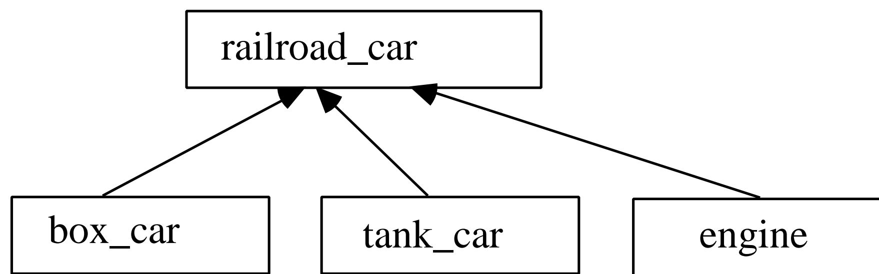

```
class gondola_car {
public:
   double volume ()
      {return 1/2*height*width*length;}
};
```

- La definizione nella sottoclasse (**`gondola_car`**) ricopre quella della superclasse

- Per riferire la funzione ricoperta (qualificazione):

  `box::volume`

Metodi virtuali e overriding




```
class railroad_car
{public:
   railroad_car() {}     //costruttore
   void display_name()
              {cout << "rrc";}
};

class engine: public railroad_car
{public:
   engine() {}     //costruttore
   void display_name()
                  {cout << "en";}
};

...
railroad_car *P;
P = new engine;
P->display_name();   //stampa rrc
```

- C++ stabilisce qual e` la funzione da utilizzare a ***tempo di compilazione***, sulla base della dichiarazione del tipo (**`railroad_car`**)

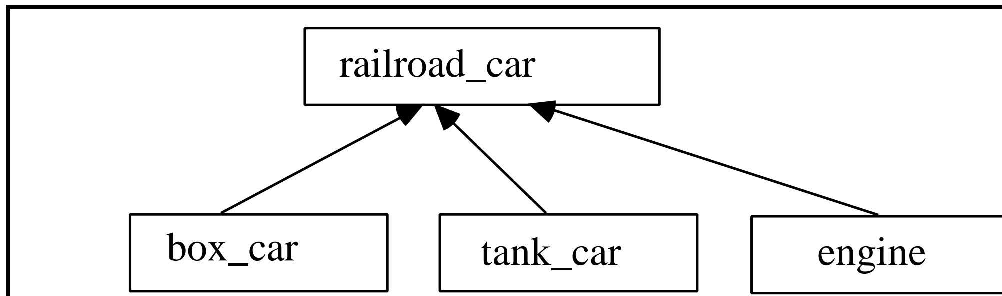
Metodi virtuali e overriding (cont.)

- Il problema sorge per la presenza di:
  - un puntatore (di tipo superclasse, **`railroad_car`**) ad un oggetto creato dinamicamente (di tipo sottoclasse, **`engine`**) e
  - dello stesso metodo (**`display_name`**) in entrambe le classi

```
engine P;
P -> display_name();   // stampa en
```

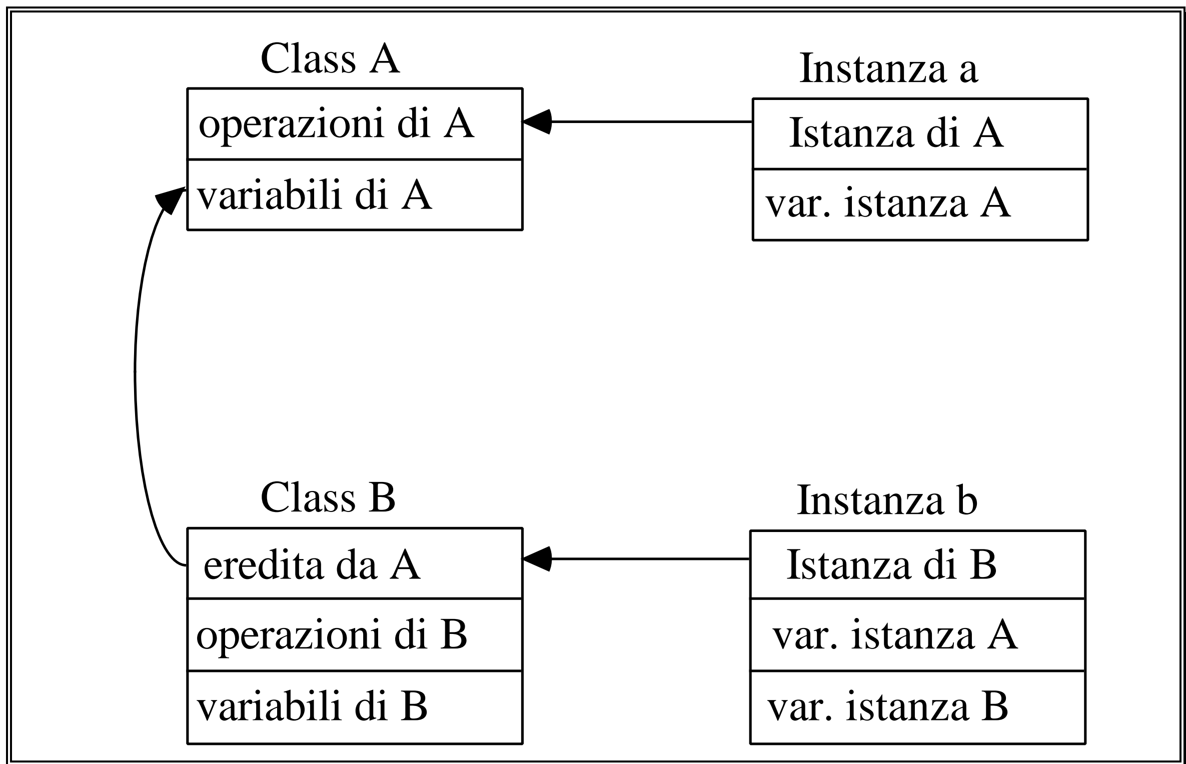
- Per stabilire dinamicamente qual e` la funzione da utilizzare, si ricorre ai ***metodi virtuali*** (l'informazione a tempo di compilazione e` solo "virtuale" e non viene utilizzata)

- I ***metodi virtuali*** sono soggetti a ridefinizione nelle sottoclassi (*binding ritardato* della chiamata)




```
class railroad_car
{public:
   railroad_car() {}      //costruttore
   virtual void display_name()
               {cout << "rrc";}
};

class engine: public railroad_car
{public:
   engine() {}     //costruttore
   virtual void display_name()
                    {cout << "en";}
};

...
railroad_car *P;
P = new engine;
P->display_name();   //stampa en
```

Ereditarieta` e self-behaviour:



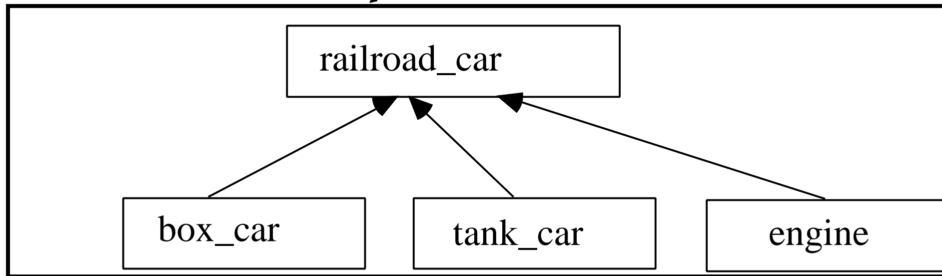
- Se b, istanza di B, non trova il metodo invocato localmente, lo "cerca" nella superclasse A

- Se lo trova in A, il metodo viene comunque eseguito sui dati di b

- Il metodo trovato deve sempre essere eseguito nell'ambiente della classe base (istanza a cui e` stato richiesto il metodo), indipendentemente dalla classe in cui e` definito il metodo (***dynamic binding***)

- A tale istanza e` legata la variabile globale ***self*** (Smalltalk) (identificatore riservato **`this`** in C++)

Metodi virtuali e self:




```
#include <iostream.h>
 class railroad_ca
 {private: int length, height, width
  public
    railroad_car()
      {length=1; height=1; width=1;}
    virtual double volume ()
        {return length*height*width;}
    virtual void  display_capacity ()
        {cout<<this->volume()<<"\n";}
 }

 class tank_car: public railroad_ca
 {private: int length, rad
  public
    tank_car() {length=1; rad=1;};
    double volume ()
      {return 3.14*rad*rad*length;}
 }

 class box_car: public railroad_ca
 {public:    box_car() {} };

 main(
 { railroad_car *P, *Q
   P = new tank_car
   Q = new box_car
```

```
P -> display_capacity()
Q -> display_capacity() }
```

- C++ consente anche di evitare di utilizzare esplicitamente il nome **`this`**

Ereditarieta` e Polimorfismo

- ***Polimorfismo:***

  Permette di utilizzare lo stesso *nome* per identificare operazioni simili che differiscono per la realizzazione

  analogie nel nome dei servizi per oggetti diversi

Polimorfismo orizzontale

La stessa invocazione può corrispondere ad esecuzioni diverse *in caso di variabili senza tipo*

```
v Print /*v variabile non tipata*/
```

Polimorfismo verticale

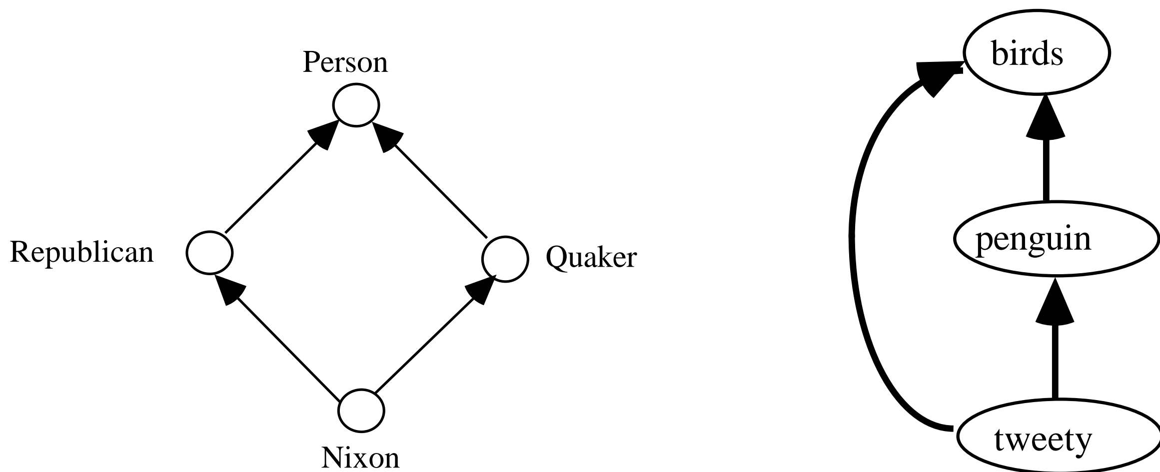
La stessa invocazio può corrispondere ad esecuzioni diverse *per ereditarietà*

```
railroad_car *P, *Q;
P = new tank_car;
Q = new box_car;
P -> display_capacity();
P = Q;
P -> display_capacity();
```

Ereditarieta` multipla

- ***Definizione:***

  Un sistema ad ereditarieta` multipla e` un sistema di ereditarieta` che non vincola i nodi ad avere grado di uscita al piu` uno



  Definita in termini di ***cammini*** in una rete (alcuni dei quali possono essere generati da archi is-a ridondanti)

```
class box_car:
   public railroad_car,
   public box{  ...};
```

Ereditarieta` multipla:

semplice	***multipla***
da una ***sola*** classe	o da ***più*** classi

- Ereditarieta` multipla puo` essere specificata come segue in C++:

```
class T:  public T1,
          public T2,...,
          public Tn;
```

- I metodi invocabili dagli oggetti della classe **T** sono specificati mediante la combinazione dei metodi ereditati da **T1**, ..., **Tn** ed i metodi definiti direttamente in **T**

- La possibilità di ***ereditare da più genitori*** rende possibile il riutilizzo di codice o attributi presenti in piu` catene di ereditarieta`, ma aumenta la possibilità di conflitti

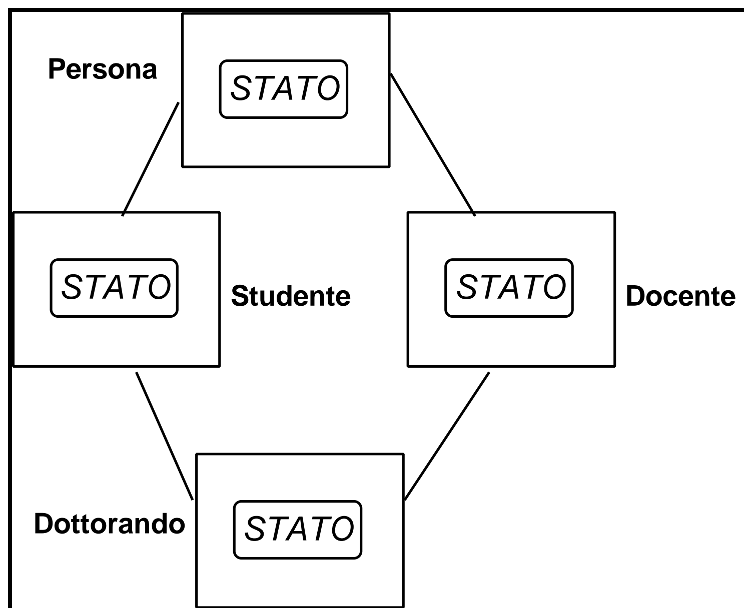
- Possibilità di conflitti a ***livello di nome***
  - sia sulle ***variabili***
  - sia sulle ***operazioni***

Ereditarieta` multipla: conflitti

- Piu` superclassi definiscono lo stesso metodo o attributo (overriding):
  - la specifica ***più vicina*** è quella visibile
  - altre sono ottenibili con notazione opportuna

- Per semplificare le regole di combinazione di variabili e metodi, alcuni sistemi (tra cui C++) rendono la gerarchia "piatta" ordinando le classi genitrici, in un ordine da sinistra a destra

- La ***linearizzazione*** puo` non essere banale nel caso di un reticolo:



Ad esempio:

Dottorando --> Studente, Docente, Persona

La classe viene pero` separata dai suoi genitori

Studente ricopre **Docente**

- Occorrono ***strategie di ricerca*** (***ereditarieta` procedurale***)

Ereditarieta` procedurale

- E` definita solo in termini di un ***algoritmo*** che opera sulla struttura dati rappresentante il grafo di ereditarieta`

Ereditarieta` procedurale

Algoritmo: *(qui definito per i soli attributi)*

Dato un oggetto A ed una proprieta` P, trovarne il valore V:

1. Se c'e` un valore per la proprieta` P in A restituisci quel valore.
2. Altrimenti, sia Class l'insieme di tutte le classi che sono puntate da archi che escono da A;
3. Se Class e` vuoto restituisci fallimento;
4. Altrimenti seleziona una classe C in Class. Se in C c'e` un valore per la proprieta` P restituisci quel valore.

Altrimenti aggiungi a Class tutte le classi (non ancora considerate) puntate da archi che escono da C.

5. Ritorna al passo 3.

- Nel caso di ereditarieta` semplice si percorre un unico cammino nella gerarchia

- Class come pila o coda: corrisponde alle due strategie di ricerca depth-first e breadth-first

- La scelta di utilizzare una politica di ricerca di tipo depth-first e` la piu` usata (***politica a black-box***)

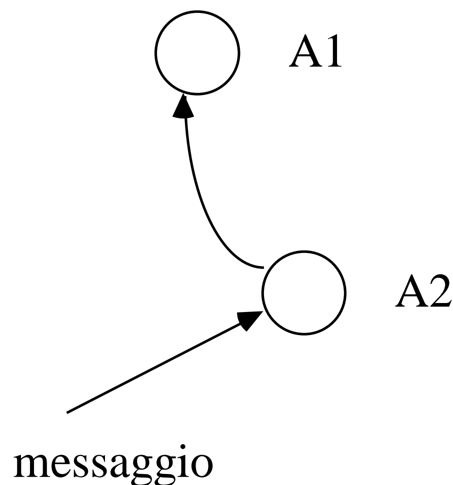
Ereditarieta` vs Delega

- Delega, introdotta nel modello ad Attori

- Ogni attore A1 puo` creare un altro attore A2, che condivide i comportamenti di A1 e puo` aggiungerne di nuovi

  A1 e` il ***prototipo*** di A2

- Quando un messaggio inviato ad A2 non puo` essere gestito localmente, A2 ***delega*** A1 a rispondere per lui



- La delega e` realizzata mediante uno ***scambio di messaggio*** esplicito fra A2 ed A1 (non implicito come nel caso del legame classe/istanza)

- Comportamento di un attore A alla ricezione di un messaggio M:

1. ***M e` un messaggio normale.***
   Se un metodo locale ad A puo` *accettare* M, A lo esegue con se stesso come *cliente*;
   Altrimenti A manda un messaggio di delega di M con se stesso come cliente ad un suo ***prototipo*** P

2. ***M e` un messaggio di delega con C come cliente***
   Se A puo` accettare M, lo esegue con C come cliente.
   Altrimenti, a sua volta invia un messaggio di delega con C come cliente ad un suo ***prototipo*** P

Smalltalk-80

Smalltalk e` il primo linguaggio con ambiente prototipale

programmazione prototipale *con esecuzione immediata senza* ***soluzione di continuità*** *tra* ***applicazione e sistema***

- semantica **per riferimento**
  - nessun innestamento di oggetti
  - *garbage collection per oggetti non più riferibili*
- variabili non **tipate**
  - polimorfismo orizzontale
  - *controlli solo dinamici di correttezza delle variabili, con ovvio overhead*
- massima **capacità espressiva**
  - possibilità di strutture molto flessibili, ad es. *liste di elementi disomogenei*
- risoluzione del **binding solo dinamica**
  - le variabili ed i metodi sono legati solo al momento delle esecuzione
  - *ogni cambiamento si manifesta in ogni azione sull'oggetto variato*
- altro: **ereditarietà semplice**

C++

C++ si allinea alla filosofia del C fornendo anche risposta ai problemi irrisolti

> *programmazione per produzione di **programmi efficienti** e **senza vincoli espressivi***

- semantica **per riferimento** (puntatori) e **per innestamento**
  - oggetti possono essere anche innestati in altri e legati al destino dei contenenti
- variabili **tipate**
  - controlli statici delle variabili
  - *possibile visibilità anche esterna delle variabili*
- risoluzione del **binding statica** e **dinamica**
  - le variabili ed i metodi sono legati prima possibile - a meno di definizioni *virtual*
- controllo **esplicito** di creazione/distruzione
  - gli oggetti - anche innestati - prevedono espliciti creatori e distruttori da invocare
  - *no garbage collector*
- altro: **ereditarietà** multipla

  genericità di classi (*template*)

Eiffel

Eiffel rende più sicura la programmazione OO

programmazione ***ben controllata*** *e* ***verificata*** *di applicazioni*

- semantica **per riferimento**
  - oggetti contengono solo riferimenti ad altri
  - uso di *attributi* (sia variabili, sia metodi)
- variabili **tipate**
  - controlli statici delle variabili
- **interfaccia** come tipo per ogni classe
  - *overriding* non implicito
- **prerequisiti** su **esecuzione/oggetti** e **eccezioni**
  - si specificano vincoli da rispettare durante l'esecuzione e azioni correttive
- risoluzione del **binding dinamica**
  - le variabili ed i metodi sono legati più tardi possibile (*late binding*)
- primitive **esplicite** di sistema per la creazione/distruzione
- altro: **genericità** di classi

Java

sviluppo di applicazioni "di rete", ***byte-code*** *e interprete*

- Entita` **interfaccia** e implementazione
- semantica **per riferimento** (senza uso esplicito di puntatori)
- variabili **tipate**
  - controlli statici delle variabili
- risoluzione del **binding dinamica**
- controllo **esplicito** di creazione e **implicito** di distruzione
  - *garbage collector* (algoritmo *mark and sweep*)
- altro: **ereditarietà** **singola** per le classi

  multipla per le interfacce

  genericità di classi (*template*)

  costrutti per parallelismo (*thread*)
- forte interrelazione con Internet

  Applet, applicazioni Java visualizzabili in un browser www

Una possibile classificazione dei sistemi ad oggetti:

[Wegner, 1990]