Politecnico
di Milano

Dalla programmazione procedurale alla programmazione ad oggetti

Approccio tradizionale alla programmazione

- modulo: *procedura, libreria, pool di dati*
- metodologia di riferimento: *Decomposizione funzionale TOP-DOWN*
  - si scompone ricorsivamente la funzionalità principale del sistema da sviluppare in funzionalità più semplici
  - si termina la scomposizione quando le funzionalità individuate sono cosi' semplici da permettere una diretta implementazione
  - si divide il lavoro di implementazione (eventualmente tra diversi programmatori) sulla base delle funzionalità individuate

Esempio di decomposizione funzionale

- Un programma per la gestione del personale
  - Calcolo delle paghe mensili
  - Adeguamento degli stipendi
  - Gestione delle assunzioni
  - ....

L'approccio TOP-DOWN è una buona idea?

- E` un modo di procedere ordinato, logico e disciplinato che permette di governare la complessità
- Più adatto per progettare algoritmi che sistemi di grosse dimensioni:
  - le funzionalità di un sistema sono soggette a frequenti cambiamenti
  - si decidono troppo presto i vincoli di attivazione tra i diversi moduli (lo sviluppatore deve valutare i *tradeoff* tra spazio e tempo troppo presto)

Un (cattivo) esempio

- Per realizzare un'applicazione gestionale, di una certa dimensione, vari programmatori sono incaricati di realizzarne parti diverse
- Alberto è incaricato di programmare l'interfaccia utente, e di definire il tipo Data per memorizzare le date.
- Bruno utilizza il tipo Data (definito da Alberto) per scrivere la parte di applicazione che gestisce paghe e contributi

Codice C di Alberto Codice C di Bruno

```
struct Data {
    int giorno;
    int mese;
    int anno;
};

void stampaData(Data d) {
  printf("%d", giorno);
  if (d.mese == 1)
      printf("Gen");
  else if (d.mese == 2)
      printf("Feb");
  …
  printf("%d", anno);
}
```

La manipolazione dei dati avviene con accesso diretto alla rappresentazione del tipo:

```
Data d;
…
d.giorno=14;
d.mese=12;
d.anno=2000;
…
if (d.giorno == 27)
      paga_stipendi();
if (d.mese == 12)
      paga_tredicesime();
 …
```

inizializzazione di un oggetto di tipo Data

uso di un oggetto di tipo data

La definizione di Data cambia!

- Dopo avere rilasciato una prima versione, Alberto modifica la rappresentazione per semplificare "stampaData"

```
struct  Data {
            int giorno;
            char mese[4];
            int anno;
};
void stampaData(Data d) {
    printf("%d", d.giorno);
    printf("%s", d.mese);
    printf("%d", d.anno);
}
```

Disastro!

- Il codice scritto da Bruno non funziona più!
- Occorre modificarlo:

```
Data d;
…
if (d.giorno == 27)
     paga_stipendi();
if (strcmp(d.mese,"Dic"))
     paga_tredicesime()
 …
```

Le modifiche sono costose

- Se la struttura dati è usata in molti punti del programma, una modifica piccola della struttura comporta tante piccole modifiche.
- Fare tante (piccole) modifiche è:
  - fonte di errori
  - lungo e costoso
  - difficile se documentazione cattiva
- La realizzazione interna (**implementazione)** delle strutture dati è una delle parti più soggette a cambiamenti (per maggiore efficienza, semplificazione codice...)!

Soluzione

- Il problema può essere risolto disciplinando o impedendo l'accesso alla realizzazione della struttura dati.
  - Bruno non deve più scrivere: (*d.mese* == 12)
  - ma anche Alberto da solo...

Come usare una Data?

- Se si impedisce a Bruno l'accesso ai campi della struttura Data, come può Bruno utilizzare le date di Alberto?

```
struct Data {
                int giorno;
                int mese;
                int anno
                };
```

- Il responsabile per Data (Alberto) deve fornire delle operazioni sufficienti agli utilizzatori o clienti del tipo Data (Bruno)

Funzioni predefinite per Data

- L'accesso alla struttura dati è fornito unicamente tramite operazioni predefinite

In C:

```
void inizializzaData(Data *d, int g, int m, int a) {
      d->giorno = g; {
      d->mese = m;
      d->anno = a;
  }

int leggi_giorno(Data d) {return(d.giorno);}

int leggi_mese(Data d){return(d.mese);}

int leggi_anno(Data d){return(d.anno);}
```

Uso delle operazioni predefinite

- Il codice di Bruno è scritto in modo tale da usare solo le operazioni predefinite:

```
Data d;
 …
inizializzaData(*d, 14,12,2000);
if (leggi_giorno(d) == 27)
   paga_stipendi();
if (leggi_mese(d) == 12)
   paga_tredicesime()
 …
```

Modifica struttura dati

- Se Alberto cambia la rappresentazione interna di Data, deve modificare anche le operazioni predefinite (ma lasciandone immutato il prototipo):

```
int leggi_mese(Data d){
    if (strcmp(d.mese,"Gen")) return(1);
  …
   else if (strcmp(d.mese,"Dic")) return(12);
   }
void inizializzaData(Data *d, int g, int m, int
  a)) {d->giorno = g;
    if (m==1) d->mese = "Gen";
  …
   }
```

- Ma il codice di Bruno non cambia!

Incapsulamento

- Abbiamo INCAPSULATO la struttura dati!
- Ogni modifica alla struttura resta confinata alla struttura stessa
- Le modifiche sono
  - più semplici
  - più veloci
  - meno costose

Tipo di dato astratto (ADT)

- Astrazione sui dati che non classifica i dati in base a loro rappresentazione (IMPLEMENTAZIONE), ma in base a loro *comportamento atteso* (SPECIFICA).
- Il comportamento dei dati è espresso in termini di un insieme di operazioni applicabili a quei dati (INTERFACCIA)
- Le operazioni dell'interfaccia sono le sole che si possono utilizzare per creare, modificare e accedere agli oggetti (Information hiding)
  - L'implementazione della struttura dati non può essere utilizzata al di fuori della definizione della struttura.

Riassumendo

- definire un **tipo di dato astratto**:
  - tipo: Data
  - valori: date definite in termini di giorno mese e anno
  - operazioni per una data D:
    - int mese( ): restituisce il mese corrispondente a D
    - ….

codice di Alberto versione 1.
```
struct Data {
 int giorno;
 int mese;
 int anno;
};

int mese(Data d){...}
```

codice di Alberto versione 2.
```
struct Data {
 int giorno;
 char mese[4];
 int anno;
};

int mese(Data d){...}
```

codice di Bruno
```
Data d;
…
if (mese(d)==12)
  paga_tredicesime()
…
```

Progettare l'interfaccia

- Interfaccia di un ADT:
  - "promessa" ai clienti del tipo che le operazioni saranno sempre valide, anche in seguito a modifiche
  - deve includere tutte le operazioni utili...
    - `giorno_dopo`: passa al giorno successivo: in C sarebbe:

```
void giorno_dopo(Data *d){
   d->giorno++;
   if (d->giorno > 31){/*ma mesi di 30, 28, 29?*/
           d->giorno = 1;
           d->mese++;
           if (d->mese > 12) {
           d->mese = 1; d->anno++;
           }
           }
   }
```

Se la funzione non fosse definita da Alberto, Bruno avrebbe difficoltà a scriverla senza accedere ai campi di una Data.

Abstract Data Type: riassunto

- ADT = Tipo definito dall'utente che incapsula tutte le operazioni che ne manipolano i valori:
  - esporta
    - il nome del tipo
    - l'interfaccia: tutte e sole le operazioni per manipolare oggetti (cioè dati) del tipo e la loro specifica
  - nasconde
    - la struttura del tipo
    - l'implementazione delle operazioni
- un cliente può creare oggetti (dati) del tipo specificato e manipolarli tramite le operazioni definite.

Linguaggi orientati agli oggetti (OO)

- Esiste un certo numero di linguaggi di programmazione ("orientati agli oggetti") che:
  - *obbligano* ad incapsulare le strutture dati all'interno di opportune dichiarazioni (CLASSI);
  - *consentono* facilmente di impedire l'accesso all'implementazione.
- Dichiarazione di un tipo di dato in Java:

```
<visibilità> class <nome-del-dato> {
        //OVERVIEW: descrizione degli oggetti
        //costruttori
        //metodi
    }
```

- Costruttori e metodi sono procedure che
  - "appartengono" agli oggetti
  - hanno come parametro implicito l'oggetto cui appartengono (*this*)

Approccio di progettazione OO (1)

- BOTTOM-UP
  - si individuano le astrazioni principali che caratterizzano il dominio applicativo e li si rappresenta nel progetto con il modulo classe
    - es. CAD: figure geometriche, triangoli, rettangoli, linee, punti, colori…
    - es. posta elettronica: messaggio, persona, indirizzario, protocollo…
    - es. applicazione gestionale: persona, impiegato, manager, consulente, progetto, stipendio, rimborso...
  - si assemblano i diversi componenti individuando i meccanismi che permettono ai diversi oggetti di collaborare tra loro per realizzare le diverse funzionalità dell'applicazione

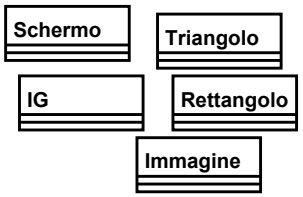
Approccio di progettazione OO (2)

- l'approccio OO permette di rappresentare le astrazioni che caratterizzano lo spazio del problema nello spazio della soluzione (implementazione) - Bruce Eckel
  - applicazioni più facili da capire e manipolare
- difficoltà di apprendimento per chi è abituato alla programmazione procedurale:
  - l'approccio OO richiede di distaccarsi profondamente dal modo di pensare procedurale
  - progettare oggetti effettivamente riutilizzabili è una attività complessa
    - l'attività principale di uno sviluppatore OO è quella di riutilizzare oggetti fatti da altri (librerie)

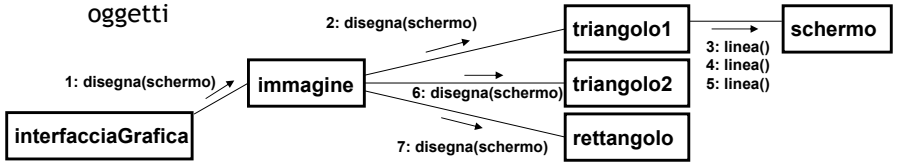
Approccio di progettazione OO (3)

come si presenta un programma:

- codice
  - insieme di definizioni di classi
  - non esiste più un grande "main()"



- configurazione run-time
  - insieme di oggetti che collaborano
  - comunicazione: invocazione delle operazioni esportate dagli oggetti

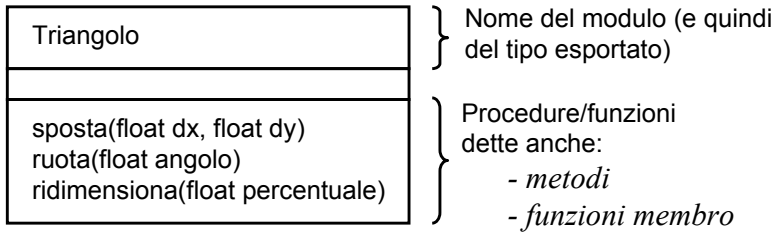


Classi (1)

- modulo che definisce un nuovo tipo di dato astratto
- nei linguaggi OO "puri" tutti i tipi sono classi (anche i tipi predefiniti come *int*, *float*, *long*…)
- nome modulo = tipo esportato
- interfaccia (parte pubblica):
  - proprietà (o variabili)
  - procedure/funzioni
- implementazione (parte privata):
  - proprietà (o variabili) - struttura dati nascosta
  - procedure/funzioni -routine di supporto alla implementazione

Classi (2)

- esempio: figura geometrica da utilizzare per l'implementazione di un CAD
- rappresentazione grafica dell'interfaccia della classe:



Classi (3)

- implementazione di una classe
  - definizione struttura dati (stato dell'oggetto)
  - implementazione dei metodi
- esempio

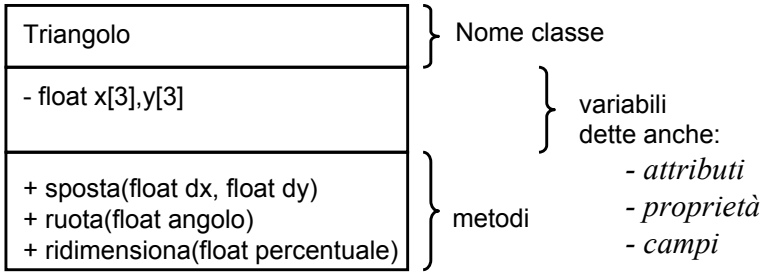
```
float x[3],y[3];

void sposta(float dx, float dy)
  {
    x[0]+=dx; x[1]+=dx; x[2]+=dx;
    y[0]+=dy; y[1]+=dy; y[2]+=dy;
  }
```

struttura dati

Implementazione del metodo "sposta"

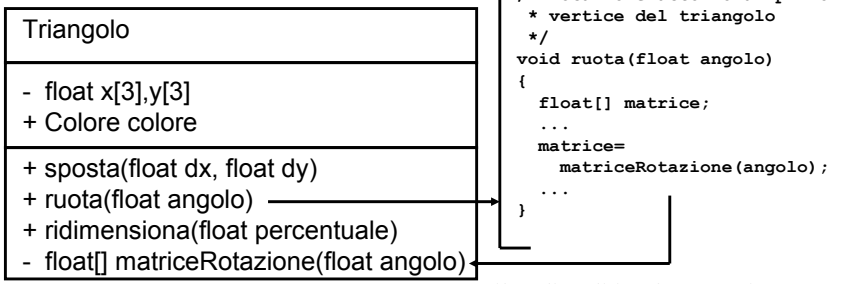
Classi (4)

- come rappresentare graficamente anche i segreti di una classe
  - tre sezioni distinte
  - ad ogni metodo e ad ogni attributo è possibile associare un simbolo che ne indica la visibilità (pubblico "+" e privato "-")
  - Il nome della classe e l'insieme dei metodi ed attributi marcati con il simbolo "+" costituiscono l'interfaccia della classe



Classi (5)

- anche gli attributi possono essere pubblici
- anche i metodi possono essere privati



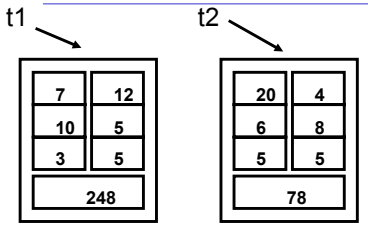
NOTA: gli attributi possono essere di qualsiasi tipo, anche classi (es. Colore)

il codice di implementazione può accedere a tutti gli attributi e a tutti i metodi della classe (anche a quelli privati)

Oggetti (1)

- le classi definiscono un tipo
- ogni oggetto è l'istanza di una classe
- ogni oggetto ha associato proprietà e procedure definite dalla classe cui appartiene
  - Le proprietà costituiscono lo stato dell'oggetto
  - Le procedure ne definiscono il comportamento
- Gli oggetti sono entità run-time; sono creati durante l'esecuzione del sistema. Le classi sono una descrizione statica (a livello di codice) dell'insieme degli oggetti di una classe.

Oggetti (2)




```
{
  Triangolo t1= ???;
  Triangolo t2= ???;

  t1.sposta(2,0);
  ...
}
```

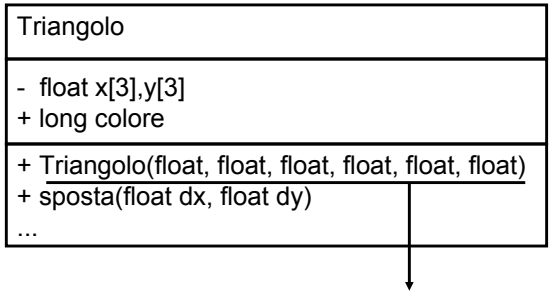
```
void sposta(float dx, float dy)
  {
    x[0]+=dx; x[1]+=dx; x[2]+=dx;
    y[0]+=dy; y[1]+=dy; y[2]+=dy;
  }
```

- invocazione di un metodo M su un oggetto X
  - il metodo agisce sullo stato dell'oggetto X
  - è come se M ricevesse come parametro anche un riferimento all'oggetto X (si veda l'implementazione di TDA in C)
  - spesso si usa l'espressione "inviare un messaggio M all'oggetto X"

Stato iniziale di un oggetto

- per definire lo stato iniziale degli oggetti, in ogni classe deve essere presente un metodo particolare detto *costruttore*
- in Java e C++ il costruttore è un metodo che ha lo stesso nome della classe
- il costruttore viene invocato automaticamente dal supporto run-time del linguaggio ogni volta che viene creato un oggetto
- il costruttore può avere dei parametri
- spesso il linguaggio fornisce un costruttore di default quando non ne è stato definito alcuno dallo sviluppatore

Costruttore



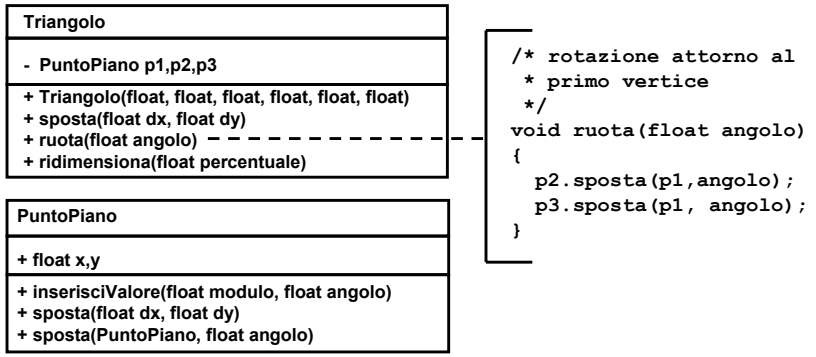

```
Triangolo(float x1, float y1, float x2, float y2, float x3, float y3){
  x[0]=x1; x[1]=x2; x[1]=x3; y[0]=y1; y[1]=y2; y[2]=y3;
  ...
}
```

Distruttore?

- il distruttore, nei linguaggi che lo prevedono, è un metodo invocato dal sistema run-time quando un oggetto viene distrutto
- funzione: liberare le risorse "occupate" dall'oggetto:
  - memoria
  - file
  - DB
  - …
- la sua presenza ed il suo ruolo dipendono fortemente dal linguaggio

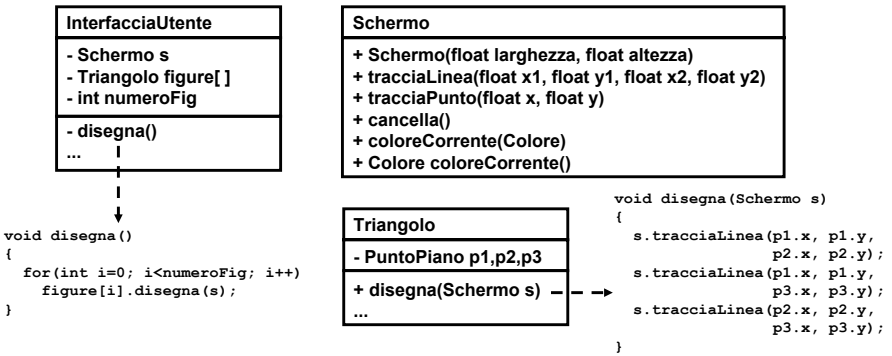
Aggregazione

- Gli oggetti possono essere incapsulati all'interno di altri oggetti



Collaborazione tra Oggetti

- gli oggetti collaborano tra loro per implementare le funzionalità del sistema
- le interfacce rappresentano il contratto che regola la collaborazione
- es. visualizzazione delle figure geometriche

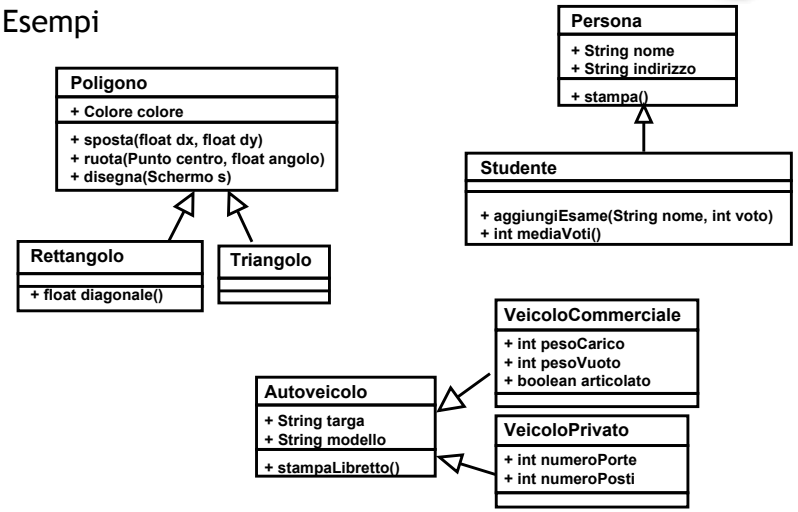


Ereditarietà (1)

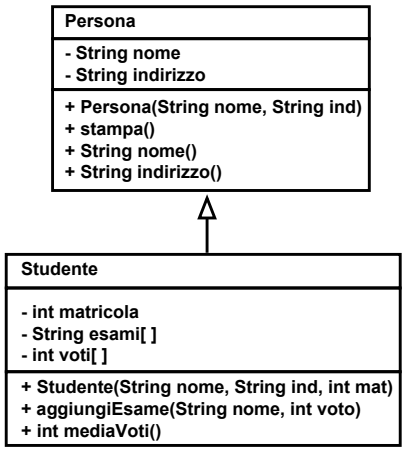
- difficoltà a riutilizzare componenti già sviluppati:
  - spesso i componenti di libreria non soddisfano pienamente le esigenza del programmatore
- il programmatore deve avere la possibilità di adattare il componente alle sue esigenza particolari SENZA alterare il componente originale
- ereditarietà
  - permette di definire una nuova classe (*classe erede*, *sottoclasse*) in funzione di una o più altre classi (*classi padre*, *super-classi*) specificandone solo le differenze
  - la classe erede è un sottotipo, una specializzazione, della (delle) classe(i) padre
  - l'ereditarietà è una gerarchia

Ereditarietà (2)

- Esempi



Ereditarietà (3)

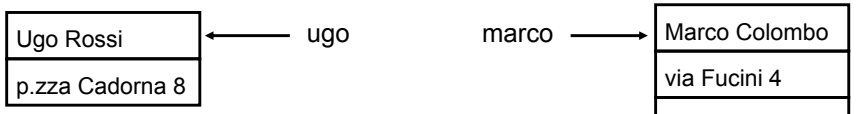


la classe Studente eredita dal padre:
- attributi
- metodi

Un oggetto Studente può essere trattato esattamente come un oggetto Persona

In cosa solitamente può differenziarsi la classe erede
- aggiunta di attributi e metodi
- i metodi possono essere ridefiniti

Ereditarietà (4)



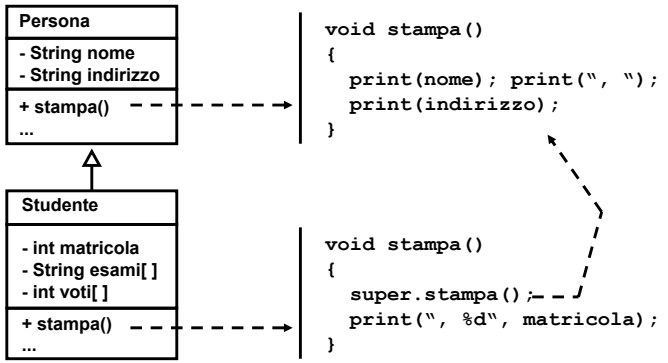

```
{
  Persona ugo("Ugo Rossi", "p.zza Cadorna 8");
  Studente marco("Marco Colombo",
                 "via Fucini 4", 53958);
  marco.aggiungiEsame("analisi 1",27);
  marco.aggiungiEsame("fisica 1",25);

  ugo.stampa(); print("\n");
  marco.stampa();
}
```

OUTPUT:

```
Ugo Rossi, p.zza Cadorna 8
Marco Colombo, via Fucini 4
```

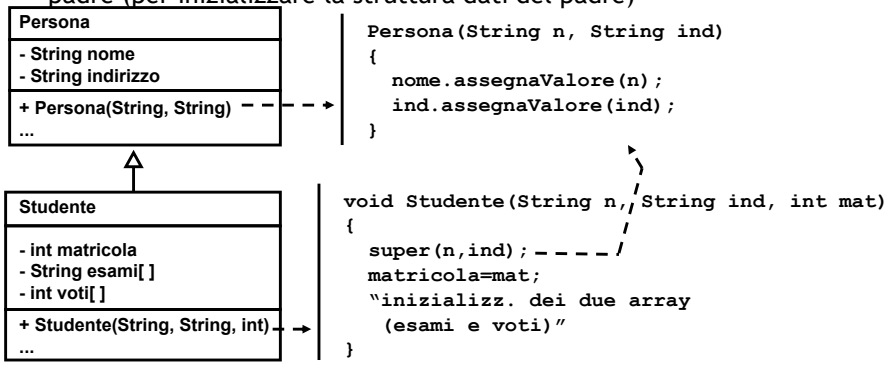
Ereditarietà (5)

- ridefinizione del metodo "stampa"
- L'implementazione dei metodi caratteristici di una classe erede (nuovi metodi o ridefinizione di metodi ereditati) può richiamare esplicitamente i metodi dei padri



Inizializzazione

- Il costruttore non viene ereditato: la sottoclasse deve sempre definirlo
- L'erede deve sempre richiamare come prima cosa il costruttore del padre (per inizializzare la struttura dati del padre)



Ereditarietà ed information hiding

- Il codice che implementa i metodi della classe erede vede i segreti dei padri?
  - in caso affermativo sarebbe violata l'information hiding
  - in caso negativo i metodi nuovi (o ridefiniti) avrebbero possibilità limitate
- La soluzione adottata in genere dai linguaggi è quella di introdurre un altro livello di visibilità e le seguenti regole:
  - public - accessibile a tutti (client, classe stessa, classi erede)
  - protected - accessibile solo alla classe stessa ed agli eredi
  - private - accessibile solo alla classe stessa

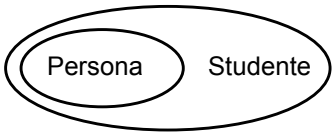
Le due interfacce di una classe

- grazie all'ereditarietà gli utilizzatori di una classe sono due:
  - i moduli "client esterni"
  - le classi erede
- a ciascun utilizzatore corrisponde una interfaccia diversa:
  - "client esterni": l'insieme dei metodi ed attributi pubblici
  - eredi: campi pubblici e protetti
- affinché un modulo sia realmente riutilizzabile è importante che entrambe le interfacce siano studiate accuratamente
- regola generale: nascondere il più possibile

Ereditarietà ed interfacce (1)

Legame tra interfaccia di una classe e quella dei propri eredi

- Le regole enunciate in precedenza prevedono che l'interfaccia del padre sia "contenuta" in quella del figlio
  - l'erede può essere trattato come il padre (viene ereditato il contratto tra classe e clienti esterni)

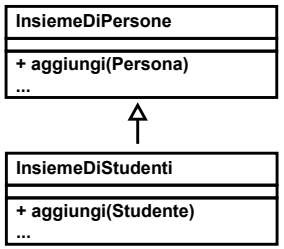


- Alcuni linguaggi seguono una filosofia differente

Ereditarietà ed interfacce (2)

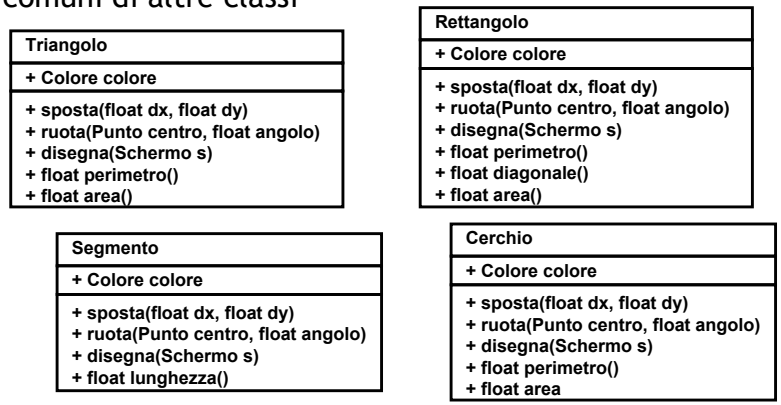
Ridefinizione dei metodi, esistono diverse politiche:

- il metodo ha la medesima interfaccia (tipo parametri e tipo del valore di ritorno) di quello che ridefinisce
- i parametri possono essere sottoclasse di quelli del metodo originale (covarianza)
- ...



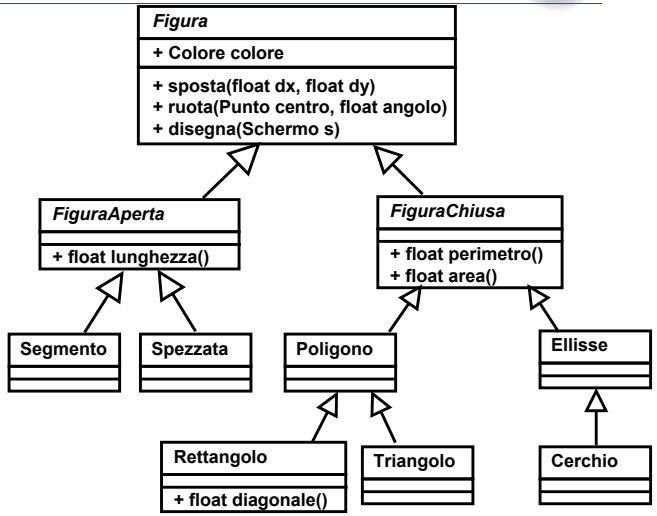
Ancora sull'ereditarietà

- l'ereditarietà promuove l'astrazione: il programmatore è indotto a specificare classi che catturano gli aspetti comuni di altre classi



Gerarchia delle figure

- esempio di classificazione dei tipi di dati astratti tramite ereditarietà



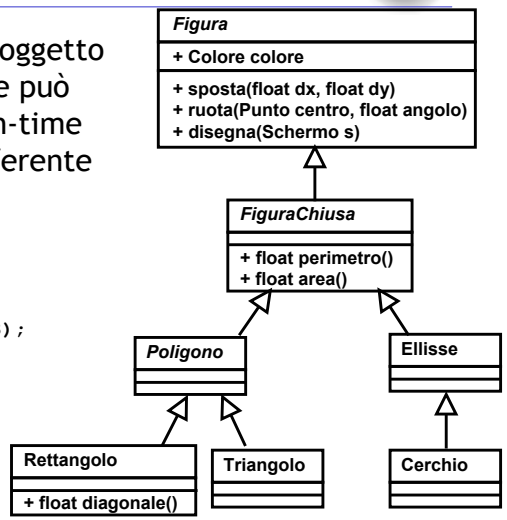
Classi astratte

- una classe è astratta quando non fornisce una completa implementazione della propria interfaccia
- una classe astratta non può essere istanziata
- le classi astratte servono come punto di partenza per la costruzione di altre classi
- una classe astratta può avere tutti i metodi non implementati
  - serve a specificare l'interfaccia comune ad un insieme di classi

Polimorfismo (1)

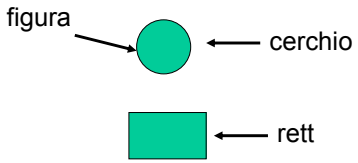
- un riferimento ad un oggetto è detto polimorfico se può essere associato a run-time ad oggetti di tipo differente
- esempio:

```
...
Figura figura;
Rettangolo rett(12,2,5,6,8,33);
Cerchio cerchio(8,9,12);

figura=cerchio;
figura.disegna(schermo);
figura=rett;
figura.disegna(schermo);
```



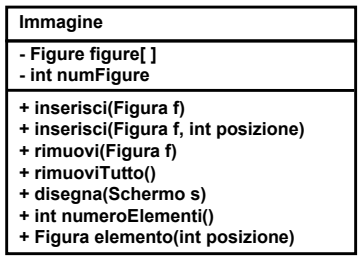
Polimorfismo (2)

```
...
Figura figura;
Rettangolo rett(12,2,5,6,8,33);
Cerchio cerchio(8,9,12);

figura=cerchio;
figura.disegna(schermo);

figura=rett;
figura.disegna(schermo);

float f=figura.diagonale();
```

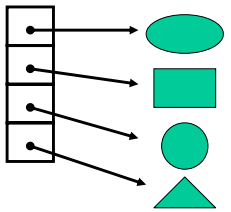


- un riferimento polimorfico ha
  - *un tipo statico*
  - *un tipo dinamico* - è il tipo dell'oggetto associato
- controllo statico dei tipi e polimorfismo
  - idea:
    - attraverso un riferimento polimorfico permettere l'invocazione di tutti i metodi definiti dal suo tipo statico
    - garantire che il tipo dinamico sia sempre coincidente con quello statico oppure una sua specializzazione (sottotipo)
  - come: controllo degli assegnamenti

Esempio polimorfismo (1)



- Immagine è una sequenza ordinata di figure
- Immagine è una struttura dati polimorfica: le sue istanze possono contenere oggetti di tipo diverso

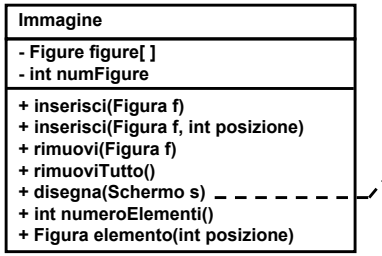


```
Cerchio c(...);
Ellisse e(...);
Rettangolo r(...);
Triangolo t(...);

Immagine imm(...);
imm.inserisci(e); imm.inserisci(r);
imm.inserisci(c); imm.inserisci(t);
```

modulo client

Esempio polimorfismo (2)

- "Immagine" non conosce i dettagli dei diversi tipi di figure (rettangoli, cerchi, triangoli…)
- La classe "Immagine" sa che TUTTE le figure sono in grado di rispondere all'invocazione dei metodi specificati nella classe "Figura"




```
void disegna(Schermo s)
{
  for(int i=0; i<numFigure; i++)
    figure[i].disegna(s);
}
```

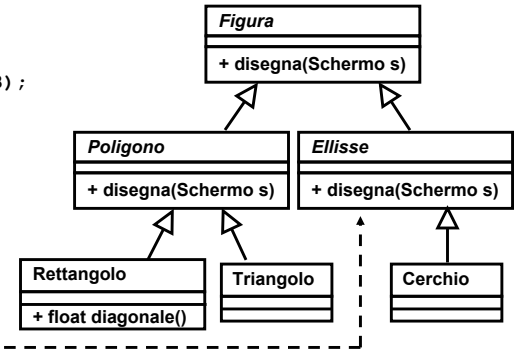
- Il metodo "disegna" funziona correttamente?

Binding dinamico (1)

- Quando viene chiamato un metodo attraverso un riferimento polimorfico, il metodo effettivamente chiamato è quello definito dal tipo dinamico

```
...
Figura figura;
Rettangolo rett(12,2,5,6,8,33);
Ellisse ellisse(8,9,12,4);

figura=ellisse;
figura.disegna(schermo);
...
```

viene chiamato il metodo "stampa" definito nella classe "Ellisse"

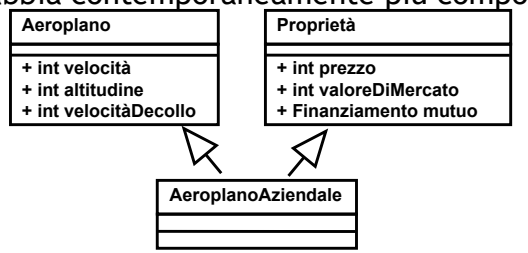


Binding dinamico (2)

- il binding è dinamico nel senso che non viene risolto dal compilatore ma a run-time dal'ambiente che supporta l'esecuzione dei programmi
- ereditarietà+polimorfismo+binding dinamico = codice generico, "facilmente" comprensibile e modificabile
  - si può interagire con gli oggetti ai diversi livelli di astrazione

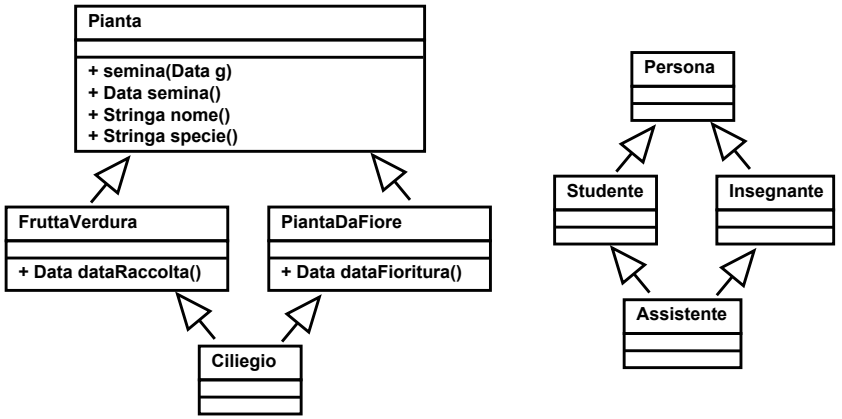
Ereditarietà multipla

- attraverso l'ereditarietà una sottoclasse eredita dalla sua superclasse:
  - contratto
  - implementazione
- l'ereditarietà multipla è utile quando si vuole che una classe abbia contemporaneamente più comportamenti



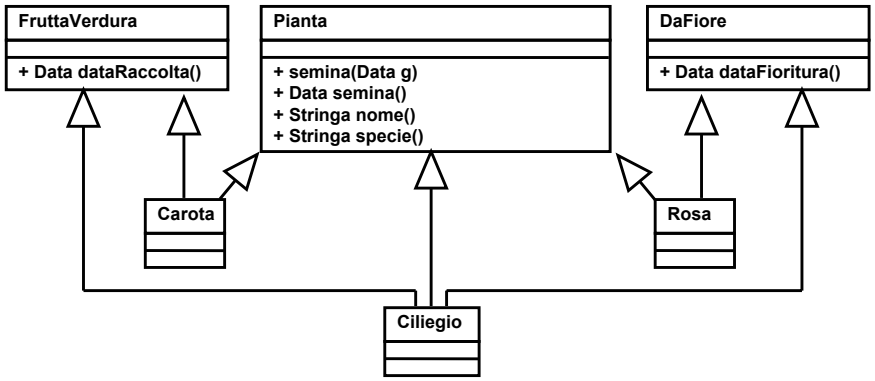
Problemi dell'ereditarietà multipla

- Le implementazioni possono entrare in conflitto



Una possibile soluzione

classi “Mix-in” (Grady Booch)



Approcci all’ereditarietà multipla

opinioni contrastanti sulla ereditarietà multipla

- complessa, misteriosa, error-prone, il goto della progr. OO
- raramente utilizzata ma in alcuni casi indispensabile
- utile e necessaria sebbene complessa

i linguaggi offrono diversi approcci:

- fornire solo l’ereditarietà semplice
  - Smalltalk
- fornire l’ereditarietà multipla e i costrutti per risolvere i conflitti
  - C++, Eiffel
- supporto all’ereditarietà multipla dei contratti ma non dell’implementazione
  - Java (interfacce)

Vantaggi derivanti dai concetti OO

- Modularità
- Information Hiding
- Estendibilità
- Integrabilità
- Riutilizzo del codice

Vantaggi derivanti dai concetti OO

Modularità

- supportata sia dalla metodologia di design che dai costrutti del linguaggio
- OOP: classe = modulo
- OOP altre proposte:
  - cluster di classi legate logicamente
  - sottosistemi di classi

Vantaggi derivanti dai concetti OO

Information Hiding

- OO: netta distinzione tra l'interfaccia di una classe e la sua implementazione
- possibili diverse implementazioni dell'interfaccia

Vantaggi derivanti dai concetti OO

Estendibilità

- ereditarietà:
  - il riutilizzo di classi esistenti facilita la definizione di nuove classi
  - tanto più l'albero di ereditarietà è ricco , tanto meno sforzo deve essere speso per sviluppare una nuova classe
- ereditarietà + polimorfismo
  - esempio

Esempio ereditarietà + polimorfismo

Soluzione tradizionale

- 3 tipi di figure, array di figure geometriche, per ogni figura calcolare l'area
- codice tradizionale:

```
while elemento
begin
  if elemento isa "cerchio" then write p X raggio X
  raggio
  elseif elemento isa "quadrato" then write lato X lato
  elseif elemento isa "triangolo" then write base X
                                          altezza X 0.5
  end
  elemento++
end
```

Esempio ereditarietà + polimorfismo

Soluzione OOP

- definizioni:
  - tre classi (cerchi, quadrati, triangoli) più una super classe (elemento)
- codice OO:

```
while elemento
begin
  write elemento.area
  elemento++
end
```

- notare:
  - il codice globalmente da scrivere non si riduce nè aumenta, vanno scritte 3 implementazioni del metodo area

Esempio ereditarietà + polimorfismo

Vantaggio rispetto all'estendibilità

- ... arrivano nuove specifiche
  - aggiungere il pentagono alle figure geometriche
- programma tradizionale
  - correggere per inserire il nuovo case
    - il rischio di correggere male è alto (devo trovare tutti i punti del codice dove un oggetto è trattato in modo diverso a seconda della forma geometrica)
    - le correzioni sono sparpagliate in tutto il sistema (vengono fatte da programmatori diversi: necessità di coordinamento o duplicazioni del lavoro)
- programma OOP:
  - si aggiunge una nuova classe e l'algoritmo funziona come prima

Vantaggi derivanti dai concetti O-O

Integrabilità

- O-O: interfacce ben definite e di limitate dimensioni
  - facilità di comprensione
  - facilità di integrazione

Vantaggi derivanti dai concetti O-O

Riutilizzo

- ogni volta che viene creata una istanza di una classe si ha riutilizzo del codice
- ereditarietà
  - riuso di una classe esistente per definire nuove classi

Un po'di storia

- Il concetto di tipo di dati astratti è dovuto a B. Liskov e S. Zilles, 1974, con il contributo di molti altri ricercatori.
- Il primo linguaggio orientato agli oggetti è stato Simula 67, definito e implementato dai norvegesi K. Nygaad e O.J. Dahl tra il '63 e il '70.
- Il primo linguaggio ad oggetti di grande successo è stato Smalltalk (Alan Kay, 1972/1980).
- C++, definito da B. Stroustrup nel 1984, e poi esteso e standardizzato da ANSI, è uno dei linguaggi più usati grazie alla sua compatibilità con il C.
- Java è stato definito dalla Sun Microsystems nel 1995/96. La versione corrente è del Febbraio 2002.