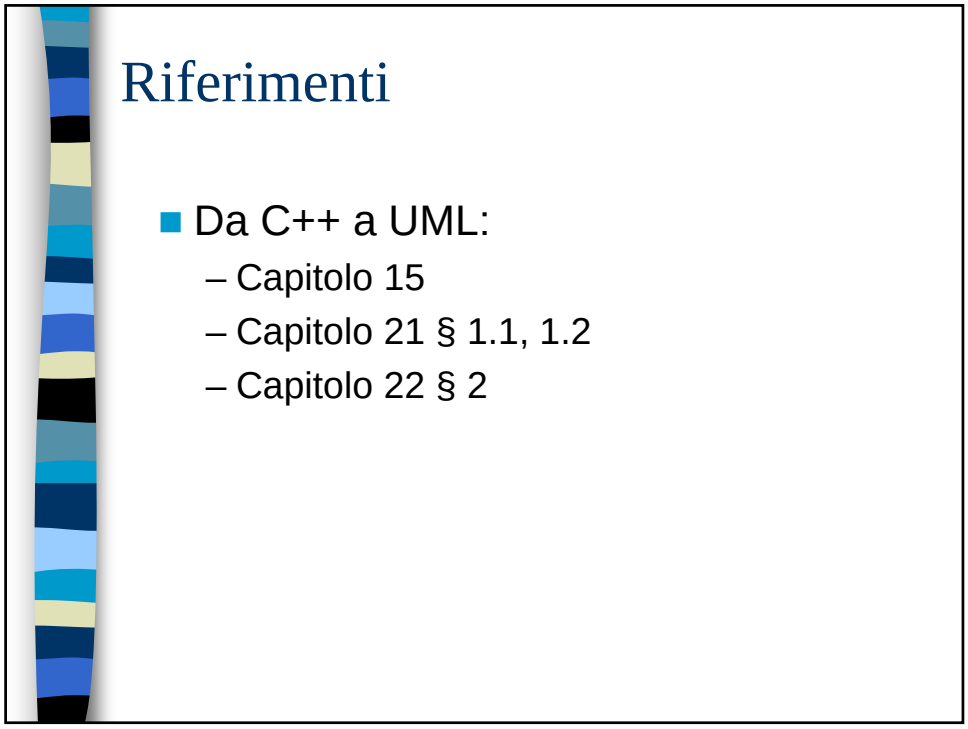
Corso di Programmazione I

La programmazione a oggetti e il C++ (introduzione)

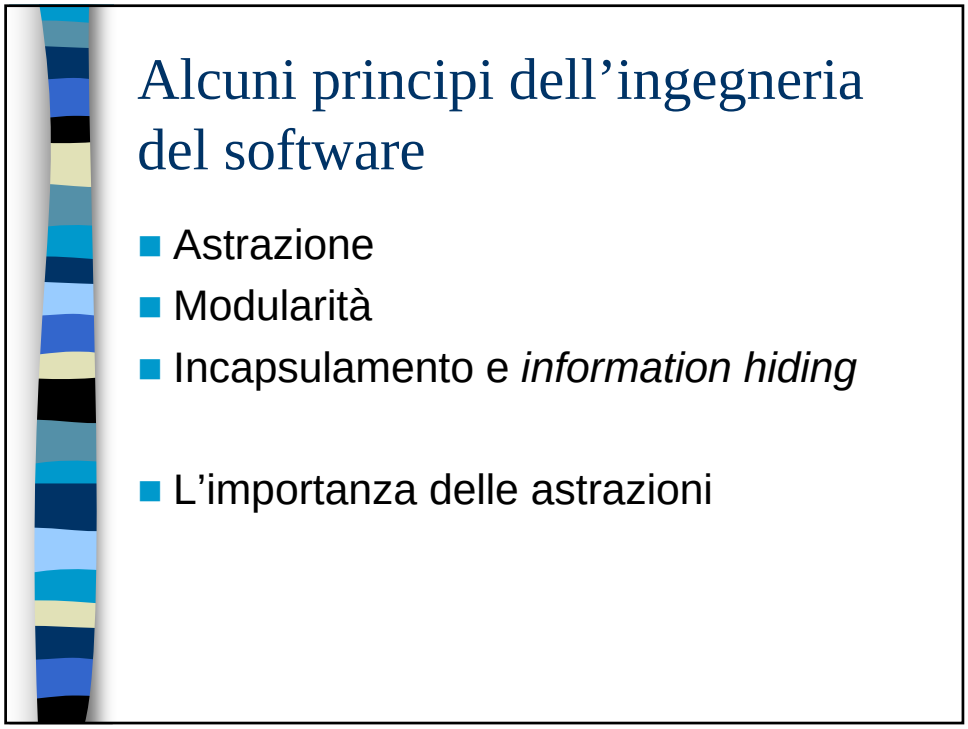
Argomenti

- I concetti di astrazione e modulo
- Meccanismi di astrazione
- Metodologie *top-down* e *bottom-up*
- Programmazione a oggetti
- Ereditarietà: concetti e motivazioni
- Polimorfismo: concetti e motivazioni
- Vantaggi della programmazione OO
- Classi in C++



Riferimenti

- Da C++ a UML:
  - Capitolo 15
  - Capitolo 21 § 1.1, 1.2
  - Capitolo 22 § 2



Alcuni principi dell'ingegneria del software

- Astrazione
- Modularità
- Incapsulamento e *information hiding*
- L'importanza delle astrazioni

L’astrazione

- L’*astrazione* è il processo che porta ad estrarre le proprietà rilevanti di un’entità, ignorando i dettagli inessenziali
  - Le proprietà estratte definiscono una vista dell’entità
  - Una stessa entità può dar luogo a viste diverse
- Esempio: un’automobile
  - vista dal venditore:
    - prezzo, durata della garanzia, colore, ...
  - vista dal meccanico:
    - tipo di motore, cilindrata, tipo di olio, ...

La modularità

- La *modularità* è l’organizzazione in parti (moduli) di un sistema, in modo che esso risulti più semplice da comprendere e manipolare
  - Gran parte dei sistemi complessi sono modulari
- Esempio: Un’automobile è suddivisa in più sottosistemi:
  - Motore
  - Trasmissione
  - ...

Il concetto di modulo

- Un modulo di un sistema software è un componente che:
  - Realizza una astrazione
  - È dotato di una chiara separazione tra:
    - *Interfaccia*
    - *Corpo*
- L'interfaccia specifica "cosa" fa il modulo (l'astrazione realizzata) e "come" si utilizza
- Il corpo descrive il "come" l'astrazione è realizzata

Incapsulamento e *information hiding*

- *L'incapsulamento* consiste nel nascondere e proteggere alcune informazioni di un'entità
- L'accesso in maniera controllata alle informazioni nascoste è possibile grazie ad un insieme di operazioni descritte dall'*interfaccia*
  - Se l'interfaccia non cambia, le informazioni nascoste possono essere modificate senza che questo influisca sulle altre parti del sistema di cui l'entità fa parte
- Esempio: un'autoradio
  - L'interfaccia consiste dei controlli e dei connettori tramite i quali è collegata all'automobile
  - I dettagli di come funziona sono nascosti
  - Per installarla e usarla non è necessario conoscere alcunché della sua struttura interna

L’importanza delle astrazioni

- *La complessità dei problemi che siamo in grado di risolvere è direttamente correlata alle specie e alle qualità delle astrazioni disponibili*
- Tutti i linguaggi di programmazione forniscono astrazioni:
  - tipi semplici
  - strutture di controllo
  - sottoprogrammi
- L’astrazione fondamentale dei linguaggi a oggetti è l’astrazione sui dati

Meccanismi di astrazione (1/2)

Nella progettazione di un sistema software è opportuno adoperare delle tecniche di astrazione per dominare la complessità del sistema da realizzare.
I meccanismi di astrazione più diffusi sono:

- ASTRAZIONE SUL CONTROLLO
- ASTRAZIONE SUI DATI

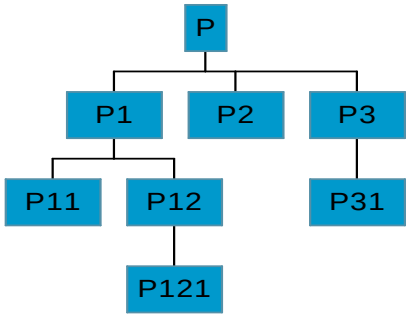
Meccanismi di astrazione (2/2)

- ASTRAZIONE SUL CONTROLLO
  - Consiste nell'astrarre una data funzionalità dai dettagli della sua implementazione;
  - E' ben supportata dai linguaggi di programmazione tradizionali tramite il concetto di sottoprogramma.
- ASTRAZIONE SUI DATI
  - Consiste nell'astrarre le entità (oggetti) costituenti il sistema, descritte in termini di una struttura dati e delle operazioni possibili su di essa;
  - Può essere realizzata con un uso opportuno delle tecniche di programmazione modulare nei linguaggi tradizionali;
  - E' supportata da appositi costrutti nei linguaggi di programmazione ad oggetti.

Metodologie di progetto: top-down e bottom-up (1/2)

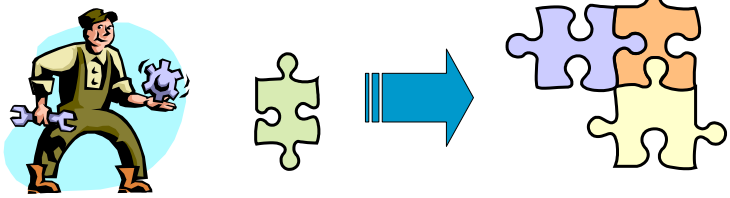
- La metodologia discendente o "**top-down**" è basata su un approccio di **decomposizione funzionale** nella definizione del sistema software, cioè sull'individuazione delle funzionalità del sistema da realizzare e su raffinamenti successivi, da iterare finché la scomposizione del sistema individua sottosistemi di complessità accettabile.

Le foglie dell'albero sono sottosistemi già esistenti (riusabili) o di complessità accettabile



Metodologie di progetto: top-down e bottom-up (2/2)

- La metodologia ascendente o "**bottom-up**" è basata sull'individuazione delle entità (classi e/o oggetti) facenti parte del sistema, delle loro proprietà e delle interrelazioni tra di esse.
- Il sistema viene costruito assemblando componenti con un approccio "dal basso verso l'alto"



La programmazione procedurale

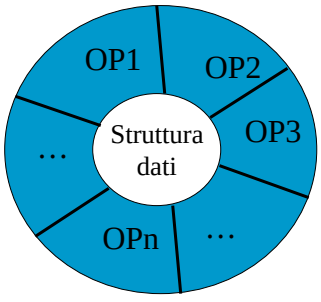
- Costituisce l'approccio tradizionale alla programmazione
- Usa come metodologia di riferimento la *decomposizione funzionale*, con approccio discendente (*top-down*)
  - si scompone ricorsivamente la funzionalità principale del sistema da sviluppare in funzionalità più semplici
  - si termina la scomposizione quando le funzionalità individuate sono così semplici da permetterne una diretta implementazione come funzioni
  - si divide il lavoro di implementazione, eventualmente tra diversi programmatori, sulla base delle funzionalità individuate

La programmazione ad oggetti

- Si individuano le *classi* di *oggetti* (entità del mondo reale o concettuale) che caratterizzano il dominio applicativo
  - le diverse classi vengono poi modellate, progettate e implementate
  - ogni classe è descritta da un'*interfaccia* che specifica il comportamento degli oggetti della classe
- L'applicazione si costruisce con l'*approccio ascendente* (*bottom-up*), assemblando oggetti e individuando le modalità con cui questi devono collaborare per realizzare le diverse funzionalità dell'applicazione

Tipi di dati astratti (1/2)

- Il concetto di tipo di dato in un linguaggio di programmazione tradizionale è quello di insieme dei valori che può assumere un dato (una variabile).
- Il tipo di dati astratto (TDA) estende questa definizione, includendo anche l'insieme di tutte e sole le operazioni possibili su dati di quel tipo. La struttura dati "concreta" è *incapsulata* nelle operazioni su di essa definite.

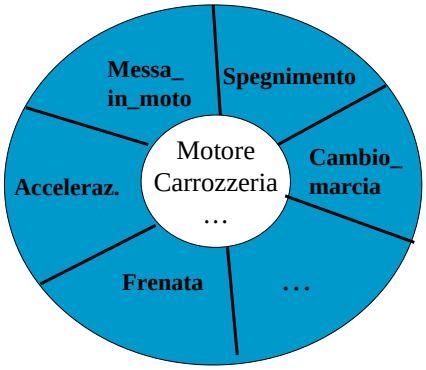


TDA

TDA (2/2)

- Non è possibile accedere alla struttura dati incapsulata (né in lettura né in scrittura) se non attraverso le operazioni definite su di essa
- Esempio:

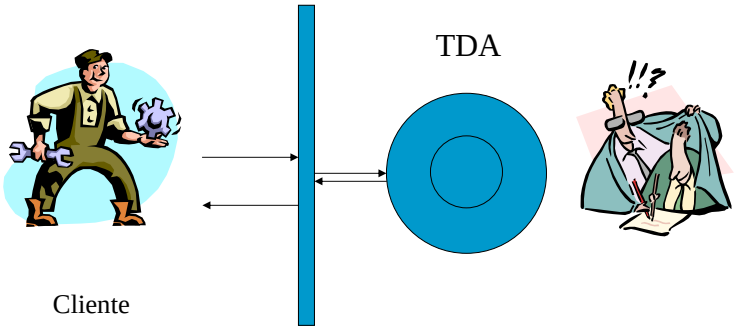
TDA Auto



- Un vantaggio: la struttura dati interna non può venire alterata da operazioni scorrette da parte dell'utente, in quanto ad essa si accede solo tramite le operazioni previste e realizzate dal produttore

Interfaccia, uso e realizzazione

- **Interfaccia**: specifica del TDA, descrive la parte direttamente accessibile dall'utilizzatore
- **Realizzazione**: implementazione del TDA



Cliente
o
Utilizzatore
del TDA

Interfaccia

Produttore
del TDA

Produttore e utilizzatore

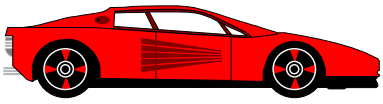
- Il cliente o utilizzatore fa uso del TDA per realizzare procedure di un'applicazione, o per costruire TDA più complessi
- Il produttore realizza le astrazioni e le funzionalità previste per il dato

→ • **Un produttore di un TDA può essere utilizzatore di un altro TDA**

- Una modifica nella sola realizzazione del TDA non influenza i moduli che ne fanno uso (in quanto non cambia l'interfaccia)

Un esempio di oggetto

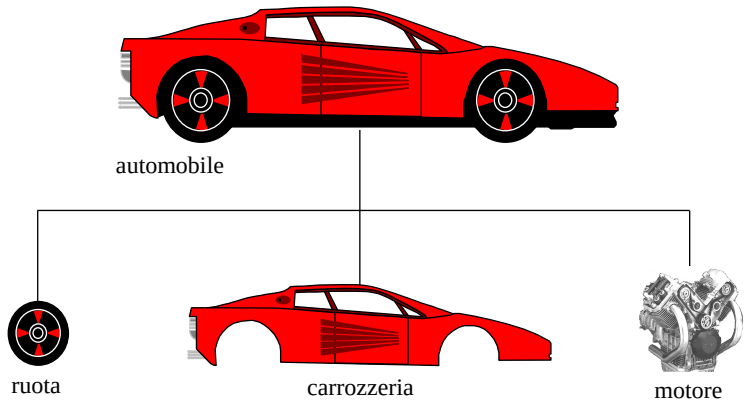
Funzioni	Dati
- Avviati	- Targa
- Fermati	- Colore
- Accelera	- Cilindrata motore
- …	- …



- Il conducente interagisce con una interfaccia per effettuare le operazioni consentite sull'automobile:
  - Pedale del freno
  - Pedale dell'acceleratore
  - Leva del cambio
  - Sistema di accensione
  - …

Oggetti composti

- Un oggetto complesso può essere composto di oggetti più semplici detti *componenti*



Tecniche di programmazione ad oggetti

- Si parla di **programmazione con oggetti** con riferimento a tecniche di programmazione basate sul concetto di oggetto (dati+operazioni)
- Si parla di **programmazione basata sugli oggetti** (*object-based programming*) con riferimento alle tecniche di programmazione basate sui concetti di:
  - Tipo di dati astratto o Classe (tipo)
  - Oggetto (istanza di un tipo)
- Si parla di **programmazione orientata agli oggetti** (*object-oriented programming, OOP*) con riferimento alle tecniche basate sui concetti:
  - Classe
  - Oggetto
  - Ereditarietà
  - Polimorfismo

Linguaggi ad oggetti (1/2)

- E' possibile adottare tecniche di programmazione con oggetti o basate sugli oggetti anche in linguaggi tradizionali (ad es., in C o Pascal), adoperando opportune discipline di programmazione, aderendo cioè ad un insieme di regole, il cui uso però non può essere forzato né verificato dal compilatore.
- Un linguaggio di programmazione ad oggetti offre costrutti espliciti per la definizione di entità (oggetti) che incapsulano una struttura dati nelle operazioni possibili su di essa.
- Alcuni linguaggi, in particolare il C++, consentono di definire tipi astratti, e quindi istanze (cioè, variabili) di un dato tipo astratto.
- In tal caso il linguaggio basato sugli oggetti presenta costrutti per la definizione di classi e di oggetti.

Linguaggi ad oggetti (2/2)

Esistono dunque linguaggi ad oggetti:

- Non tipizzati
  - Es.: Smalltalk
  - E' possibile definire oggetti senza dichiarare il loro tipo
  - In tali linguaggi, gli oggetti sono entità che incapsulano una struttura dati nelle operazioni possibili su di essa
- Tipizzati
  - Es.: C++, Java
  - E' possibile definire tipi di dati astratti e istanziarli
  - Gli oggetti devono appartenere ad un tipo (astratto)
  - In tali linguaggi, una **classe** è una implementazione di un tipo di dati astratto. Un **oggetto** è una istanza di una classe

Il linguaggio C++

- C++ è un linguaggio di programmazione *general-purpose* che supporta:
  - la *programmazione procedurale* (è un C "migliore")
  - la *programmazione orientata agli oggetti*
  - la *programmazione generica*
- C++ è quindi un linguaggio *ibrido*, nel senso che supporta più *paradigmi di programmazione*

Classi e oggetti

- Nei linguaggi a oggetti, il costrutto classe consente di definire nuovi tipi di dato (astratti) e le relative operazioni
- Sotto forma di operatori o di funzioni (dette **metodi** o **funzioni membro**), i nuovi tipi di dato possono essere gestiti quasi allo stesso modo dei tipi predefiniti del linguaggio
  - si possono creare istanze, e
  - si possono eseguire operazioni su di esse
- Un oggetto è una variabile istanza di una classe
- Lo **stato di un oggetto** è rappresentato dai valori correnti delle variabili che costituiscono la struttura dati concreta sottostante il tipo astratto

Ereditarietà e polimorfismo

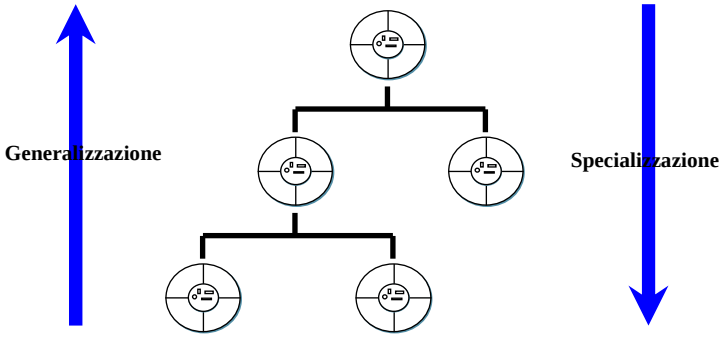
- L'**ereditarietà** consente di definire nuove classi per specializzazione o estensione di classi preesistenti, in modo incrementale
- Il **polimorfismo** consente di invocare operazioni su un oggetto, pur non essendo nota a tempo di compilazione la classe cui fa riferimento l'oggetto stesso

Ereditarietà (1/5)

- Il meccanismo dell'ereditarietà è di fondamentale importanza nella programmazione ad oggetti, in quanto induce una strutturazione gerarchica nel sistema software da costruire
- L'ereditarietà consente infatti di realizzare relazioni tra classi di tipo generalizzazione-specializzazione, in cui una classe, detta base, realizza un comportamento generale comune ad un insieme di entità, mentre le classi derivate (sottoclassi) realizzano comportamenti specializzati rispetto a quelli della classe base
- Esempio:
  - Tipo o classe base: Animale
  - Tipi derivati (sottoclassi): Cane, Gatto, Cavallo, …
- In una gerarchia gen-spec, le classi derivate sono specializzazioni (cioè casi particolari) della classe base

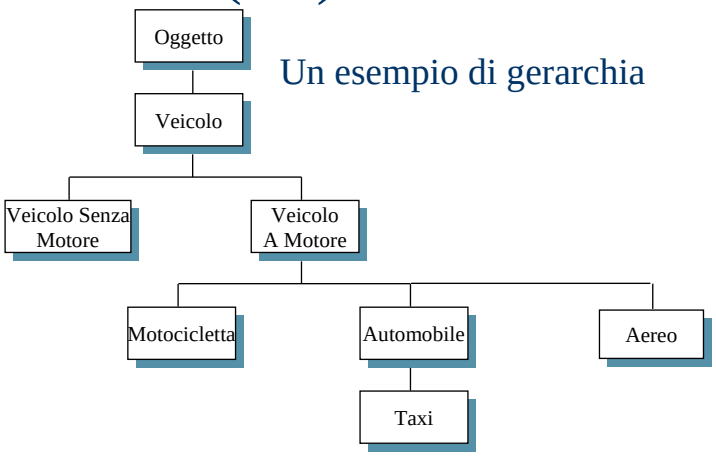
Ereditarietà (2/5)

- **Generalizzazione**: dal particolare al generale
- **Specializzazione** o particolarizzazione: dal generale al particolare



Nel paradigma a oggetti, col meccanismo dell'ereditarietà ci si concentra sulla creazione di tassonomie del sistema in esame

Ereditarietà (3/5)



Un esempio di gerarchia

- Per descrivere un sistema sono possibili tassonomie diverse, a seconda degli obiettivi

Automobile
  Benzina
  Diesel

Automobile
  Berlina
  Station Wagon

Ereditarietà (4/5)

- Esiste però anche un altro motivo, di ordine pratico, per cui conviene usare l'ereditarietà, oltre quello di descrivere un sistema secondo un modello gerarchico; questo secondo motivo è legato esclusivamente al concetto di riuso del software
- In alcuni casi si ha a disposizione una classe che non corrisponde esattamente alle proprie esigenze. Anziché scartare del tutto il codice esistente e riscriverlo, si può seguire con l'ereditarietà un approccio diverso, costruendo una nuova classe che eredita il comportamento di quella esistente, salvo che per i cambiamenti che si ritiene necessario apportare
- Tali cambiamenti possono riguardare sia l'aggiunta di nuove funzionalità che la modifica di quelle esistenti

Ereditarietà (5/5)

- In definitiva, l'ereditarietà offre il vantaggio di ridurre i tempi di sviluppo, in quanto minimizza la quantità di codice da scrivere quando occorre:
  - definire un nuovo tipo d'utente che è un sottotipo di un tipo già disponibile, oppure
  - adattare una classe esistente alle proprie esigenze
- Non è necessario conoscere in dettaglio il funzionamento del codice da riutilizzare, ma è sufficiente modificare (mediante aggiunta o specializzazione) la parte di interesse

Polimorfismo (1/4)

- Per *polimorfismo* si intende la proprietà di una entità di assumere forme diverse nel tempo.
- Una entità è polimorfa se può fare riferimento, nel tempo, a classi diverse.
- Siano ad esempio a, b due oggetti appartenenti rispettivamente alle classi A, B, che prevedono entrambe una operazione m, con diverse implementazioni. Si consideri l'assegnazione:

  a := b
- L'esecuzione della operazione m sull'oggetto a dopo l'assegnazione, per la quale è spesso adoperata la sintassi:

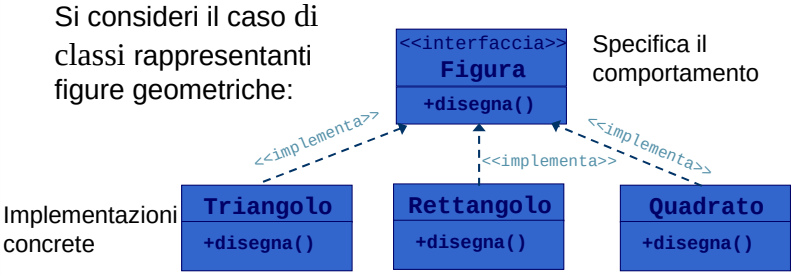
  a.m()

  produce l'esecuzione della implementazione di m specificata per la classe B.

Polimorfismo (2/4)

- Esempio:

  Si consideri il caso di classi rappresentanti figure geometriche:



- Sia ad es. A un vettore di N Figure, composto di oggetti delle classi Triangolo, Rettangolo, Quadrato:

  Figura A[N]

  (ad es.: A[0] è un quadrato, A[1] un triangolo, A[2] un rettangolo, etc.)

Polimorfismo (3/4)

- Si consideri una funzione Disegna_Figure(), che contiene il seguente ciclo di istruzioni:

```
for i = 1 to N do
        A[i].disegna()
```

- L'esecuzione del ciclo richiede che sia possibile determinare dinamicamente (cioè a tempo d'esecuzione) l'implementazione della operazione *disegna()* da eseguire, in funzione del tipo corrente dell'oggetto A[i].
- L'istruzione A[i].disegna() non ha bisogno di essere modificata in conseguenza dell'aggiunta di una nuova sottoclasse di Figura (ad es.: Cerchio), anche se tale sottoclasse non era stata neppure prevista all'atto della stesura della funzione Disegna_Figure(). (Si confronti con il caso dell'uso di una istruzione *case* nella programmazione tradizionale).

Polimorfismo (4/4)

- Il polimorfismo supporta dunque la proprietà di **estensibilità** di un sistema, nel senso che minimizza la quantità di codice che occorre modificare quando si estende il sistema, cioè si introducono nuove classi e nuove funzionalità.
- Un meccanismo con cui viene realizzato il polimorfismo è quello del *binding* dinamico.
- Il ***binding* dinamico** (*o late binding*) consiste nel determinare a tempo d'esecuzione, anziché a tempo di compilazione, il corpo del metodo da invocare su un dato oggetto.

Vantaggi della programmazione OO

- Rispetto alla programmazione tradizionale, la programmazione orientata agli oggetti (OOP) offre vantaggi in termini di:
  - ✓ **modularità**: le classi sono i moduli del sistema software;
  - ✓ **coesione dei moduli**: una classe è un componente software ben coeso in quanto rappresentazione di una unica entità;
  - ✓ **disaccoppiamento dei moduli**: gli oggetti hanno un alto grado di disaccoppiamento in quanto i metodi operano sulla struttura dati interna ad un oggetto; il sistema complessivo viene costruito componendo operazioni sugli oggetti;
  - ✓ ***information hiding***: sia le strutture dati che gli algoritmi possono essere nascosti alla visibilità dall'esterno di un oggetto;
  - ✓ **riuso**: l'ereditarietà consente di riusare la definizione di una classe nel definire nuove (sotto)classi; inoltre è possibile costruire librerie di classi raggruppate per tipologia di applicazioni;
  - ✓ **estensibilità**: il polimorfismo agevola l'aggiunta di nuove funzionalità, minimizzando le modifiche necessarie al sistema esistente quando si vuole estenderlo.

Le classi in C++

- Il linguaggio C++ supporta esplicitamente la dichiarazione e la definizione di tipi astratti da parte dell'utente mediante il costrutto **class**; le istanze di una classe vengono dette **oggetti**.
- In una dichiarazione **class** occorre specificare sia la struttura dati che le operazioni consentite su di essa. Una classe possiede, in generale, una sezione pubblica ed una privata.
- La **sezione pubblica** contiene tipicamente le operazioni (dette anche ***metodi***) consentite ad un utilizzatore della classe. Esse sono *tutte e sole le* operazioni che un utente può eseguire, in maniera esplicita od implicita, sugli oggetti.
- La **sezione privata** comprende le strutture dati e le operazioni che si vogliono rendere inaccessibili dall'esterno.
- Esempio di interfaccia di un tipo astratto Contatore in C++:

```
class Contatore {
  public:
    void Incrementa();          // operazione incremento
    void Decrementa();          // operazione decremento
  private:
    unsigned int value;         // valore corrente
    const unsigned int max;     // valore massimo
};
```

Caratteristiche di una classe

- La classe è un modulo software con le seguenti caratteristiche:
  - E' dotata di un'interfaccia (specifica) e di un corpo (implementazione)
  - La struttura dati "concreta" di un oggetto della classe, e gli algoritmi che ne realizzano le operazioni, sono *tenuti nascosti* all'interno del modulo che implementa la classe
  - Lo stato di un oggetto evolve unicamente in relazione alle operazioni ad esso applicate
  - Le operazioni sono utilizzabili con modalità che prescindono completamente dagli aspetti implementativi; in tal modo è possibile modificare gli algoritmi utilizzati senza modificare l'interfaccia

Produzione e uso di una classe

- Il meccanismo delle classi è orientato specificamente alla riusabilità del software
- Occorre dunque fare riferimento ad una situazione di produzione del software nella quale operano:
  - Il produttore della classe, il quale mette a punto
    - la *specifica*, in un file di intestazione (*header file*)
    - l'*implementazione* della classe (in un file separato)
  - L'utente della classe, il quale ha a disposizione la specifica della classe, crea oggetti e li utilizza nel proprio modulo

Utente.cpp

```
// Modulo utilizzatore del modulo
// Contatore
#include "Contatore.h"
```

Contatore.h

```
// Interfaccia del
// modulo Contatore
class Contatore {
  …
};
```

Contatore.cpp

```
// Implementazione del modulo Contatore
#include "Contatore.h"
```