

Corso di Laurea in Ingegneria Informatica

Corso di Ingegneria del Software

L'orientamento agli oggetti
(object orientation)

Sommario

- Tipi di dati astratti
- Modellazione ad oggetti
- Classi ed oggetti
- Ereditarietà
- Polimorfismo
- Vantaggi dell'O-O

Modularità

Nella progettazione di un sistema software è opportuno basarsi sul principio della **modularità**, perché esso risulti più semplice da comprendere e manipolare.

I moduli sono componenti (parti) del sistema che realizzano un'astrazione

I tipi di astrazione più diffusi sono:

- **ASTRAZIONE SUL CONTROLLO**
- **ASTRAZIONE SUI DATI**

Tipi di astrazione

ASTRAZIONE SUL CONTROLLO

- ✚ Astrazione di una **funzionalità** dai dettagli dell'implementazione
- ✚ È 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' alla base della **Modellazione a oggetti**
- ✚ E' supportata da appositi costrutti nei linguaggi di **programmazione ad oggetti**.

Astrazione sui dati

♦ Oggetti (**astrazione sui dati**):

- *Strutture dati + operazioni*
- Rispetto alle funzioni/procedure (e librerie di funzioni), hanno una struttura dati permanente, visibile solo alle funzioni interne
- La struttura dati fornisce loro uno **stato**

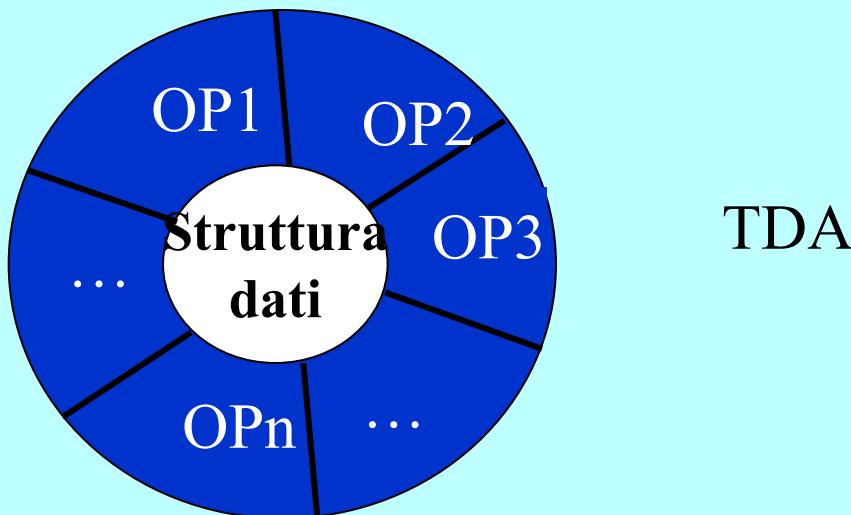
♦ Tipi di dati astratti

- Insieme alle operazioni possibili sui dati, esporta un **tipo**
- Gli esemplari del tipo sono **oggetti**

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.



Tipi di dati astratti (2/2)

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

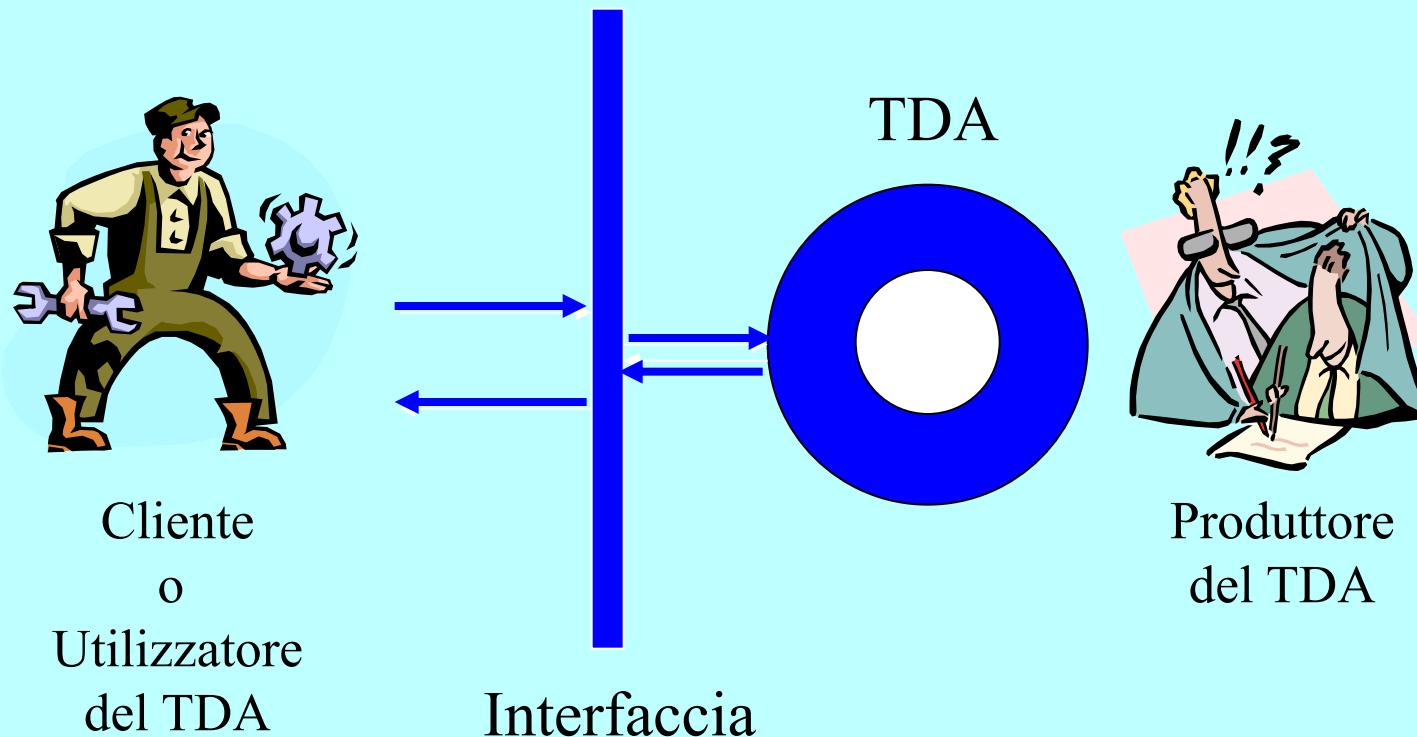
Esempio: TDA Auto



Un vantaggio: la struttura dati interna non può essere 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



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

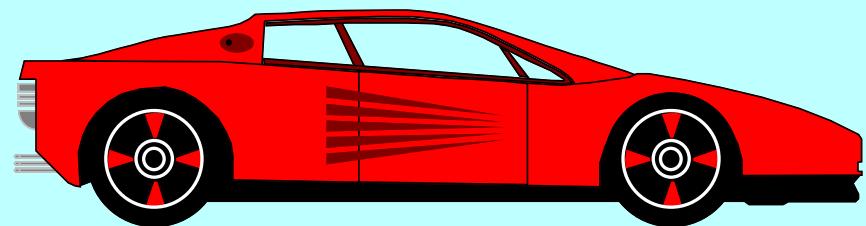
Un produttore di un TDA può essere utilizzatore di un altro TDA

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
- Accelerata	- Cilindrata motore
- ...	- ...

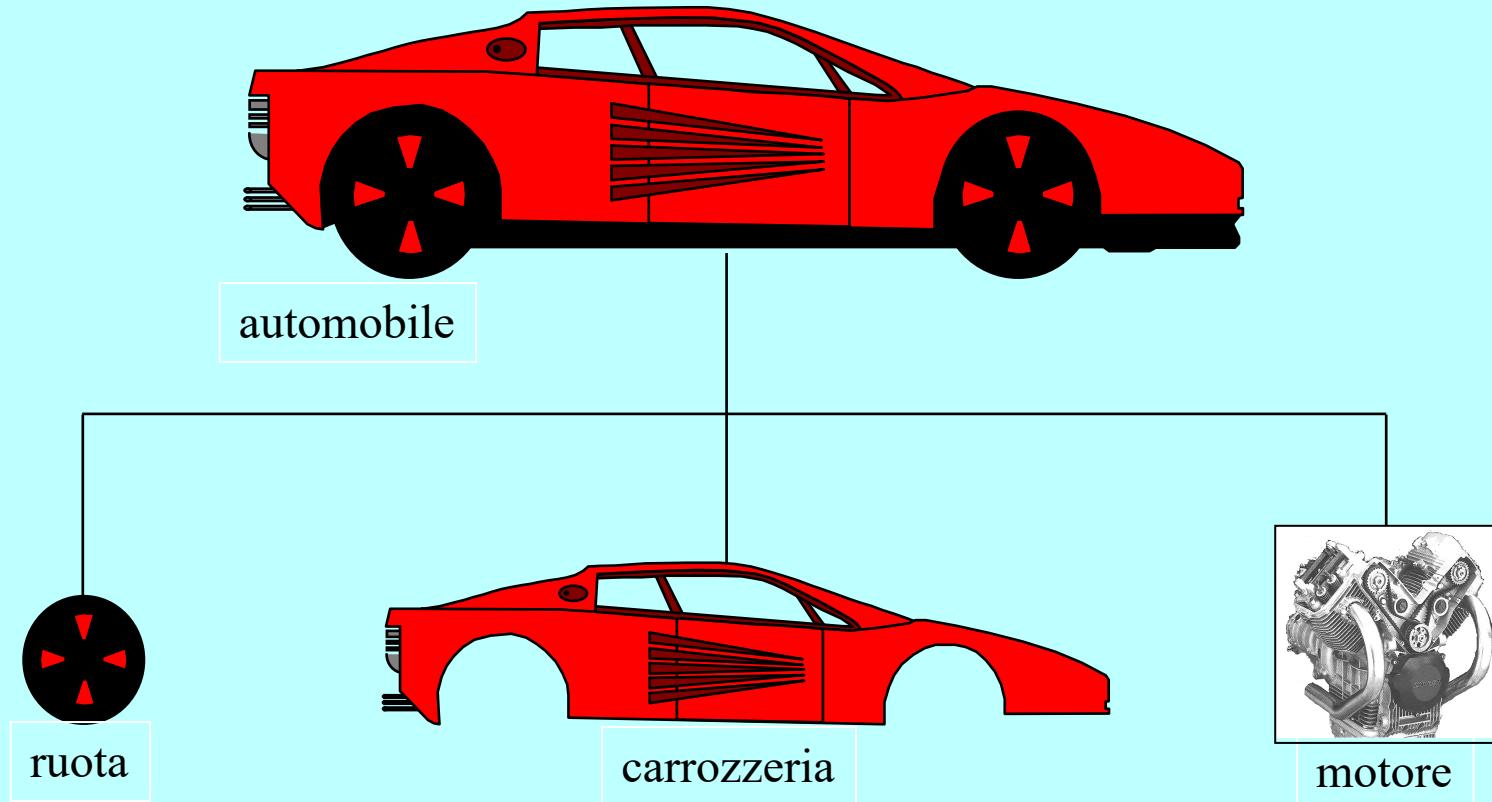


Il conducente interagisce con una interfaccia per effettuare le operazioni consentite sull'automobile:

- Pedale del freno
- Pedale dell'acceleratore
- Leva del cambio
- Sistema di accensione
- ...

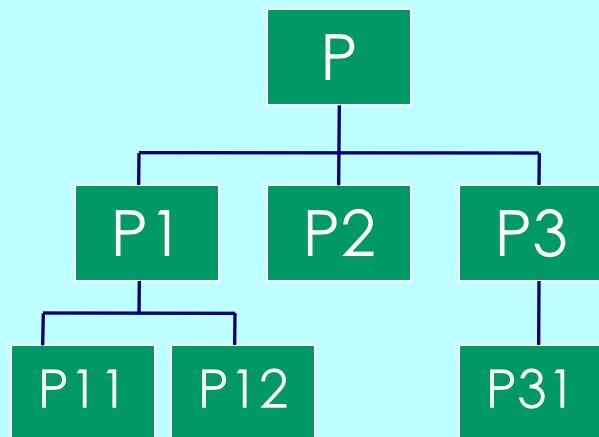
Oggetti composti

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



Metodologie di progetto

Metodologia discendente o “**top-down**”: basata sulla **decomposizione funzionale** nella definizione del sistema software => individuazione delle funzionalità e raffinamenti successivi



Metodologia ascendente o “**bottom-up**”: basata su individuazione di entità del sistema (classi e/o oggetti), delle loro proprietà e delle interrelazioni tra di esse.

Il sistema viene costruito assemblando componenti con un approccio “dal basso verso l’alto”

La modellazione a oggetti è di tipo *bottom-up*

La modellazione 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 assemblando oggetti e individuando le modalità con cui questi devono collaborare per realizzare le diverse funzionalità dell'applicazione

I principi della modellazione ad oggetti non sono vincolati ad una fase, ma riguardano l'analisi, la progettazione e l'implementazione.

La programmazione ad oggetti

A livello di implementazione, distinguiamo tra:

- ◆ **Programmazione con oggetti** con riferimento a tecniche di programmazione basate sul concetto di oggetto (dati+operazioni)
- ◆ **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)
- ◆ **Programmazione orientata agli oggetti** (object-oriented programming, OOP) con riferimento alle tecniche basate sui concetti di:
 - Classe ed Oggetto
 - Ereditarietà, Polimorfismo

Linguaggi ad oggetti (1/2)

- ◆ In linguaggi procedurali (ad es., in C) è possibile adottare tecniche di programmazione con oggetti o basate sugli oggetti, 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 esplicativi per la definizione di entità (oggetti)** che encapsulano una struttura dati nelle operazioni possibili su di essa.
- ◆ I linguaggi ad oggetti C++ e Java, consentono di definire tipi astratti (con il costrutto class), e quindi istanze (oggetti) di tipi di dato 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 linguaggi ad oggetti:
- ◆ **Non tipizzati**
 - Es.: Smalltalk
 - È possibile definire oggetti senza dichiarare il loro tipo
 - In tali linguaggi, gli oggetti sono entità che encapsulano una struttura dati nelle operazioni possibili su di essa
- ◆ **Tipizzati**
 - Es.: C++, Java
 - È 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 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

Le classi

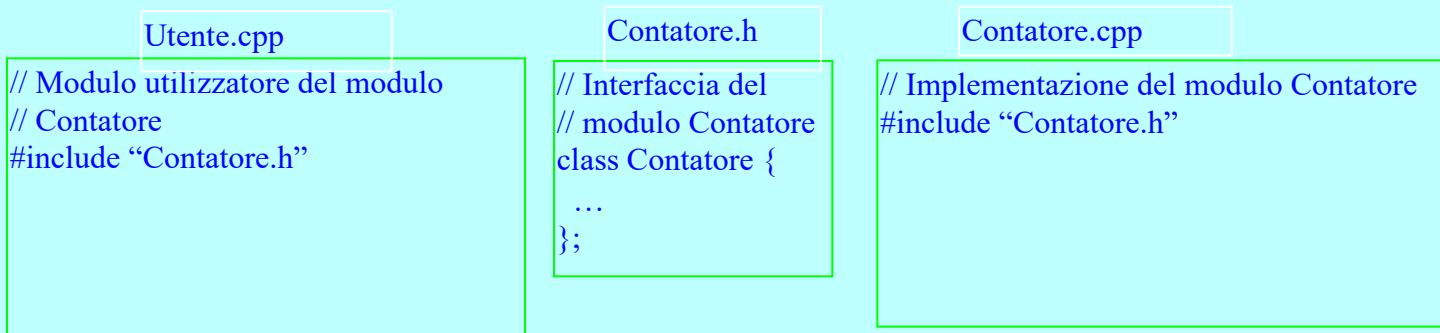
- ♦ In **fase di analisi**, le classi e le relazioni tra esse rappresentano una vista concettuale delle entità nel dominio del problema. Catturano i concetti principali del dominio, svincolandoli da come essi saranno rappresentati dal software.
 - L'attenzione dovrà essere rivolta alla comprensione di “**cosa**” il sistema software dovrà fare e non “**come**” lo fa.
- ♦ In **fase di progetto**, le classi e le relazioni tra esse formano l'architettura del sistema (ancora slegata dalla implementazione in uno specifico linguaggio)
- ♦ In **fase di implementazione**, le classi e le relazioni indicano la reale struttura del software.

Le classi in fase di progettazione

- ◆ La classe è un modulo software con le seguenti caratteristiche:
 - È 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 su di esso invocate
 - 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

- ◆ L'uso delle classi è orientato 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'utilizzatore della classe, il quale ne ha a disposizione la specifica, crea oggetti e li utilizza nel proprio modulo



Classi e Oggetti in Fase di Implementazione

- ◆ Nei linguaggi a oggetti, il **costrutto class** 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

Le classi in C++ (1/2)

- ❖ 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 sono 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.

Le classi in C++ (2/2)

- ◆ 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  
};
```

Ereditarietà e polimorfismo

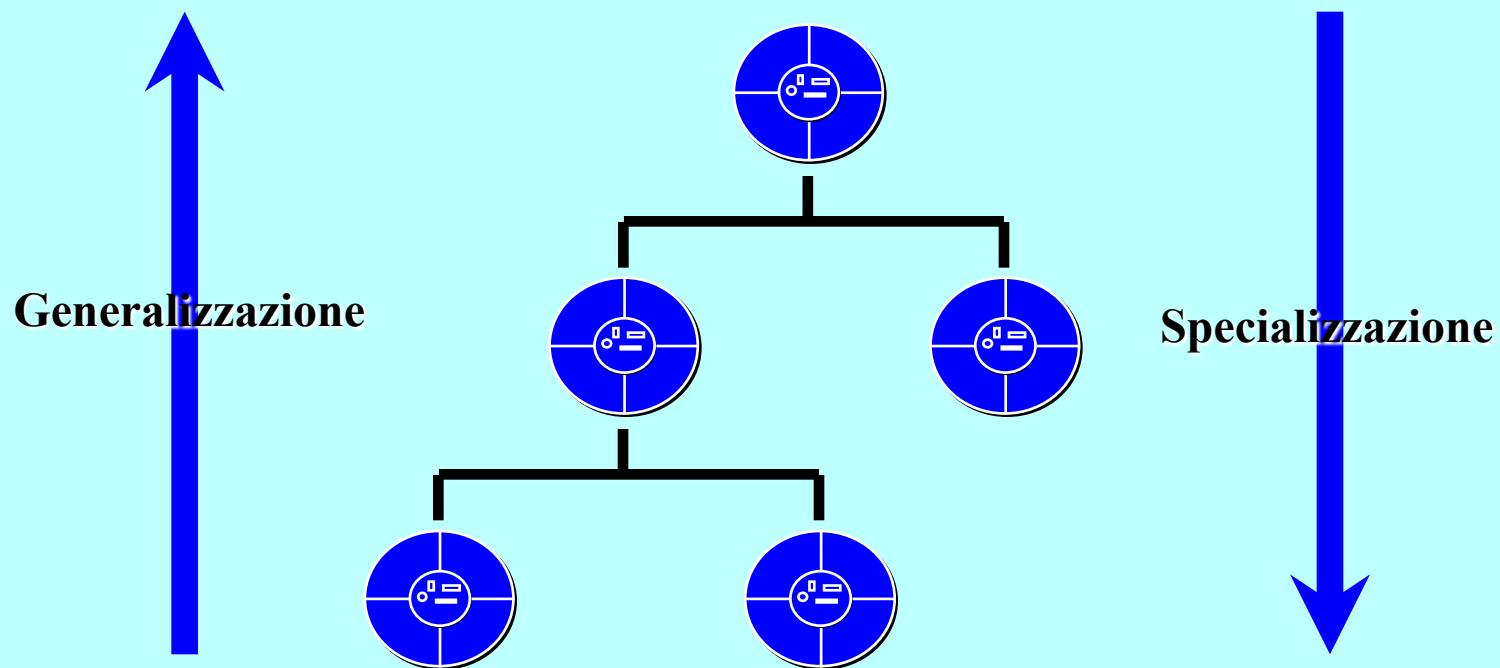
- ◆ L'**ereditarietà** consente di definire nuove classi per specializzazione o estensione di classi preesistenti, in modo incrementale
- ◆ Il **polimorfismo** consente di invocare operazioni su un oggetto, pur non essendo noto a tempo di compilazione il tipo (la classe) corrente dell'oggetto stesso

Ereditarietà (1/5)

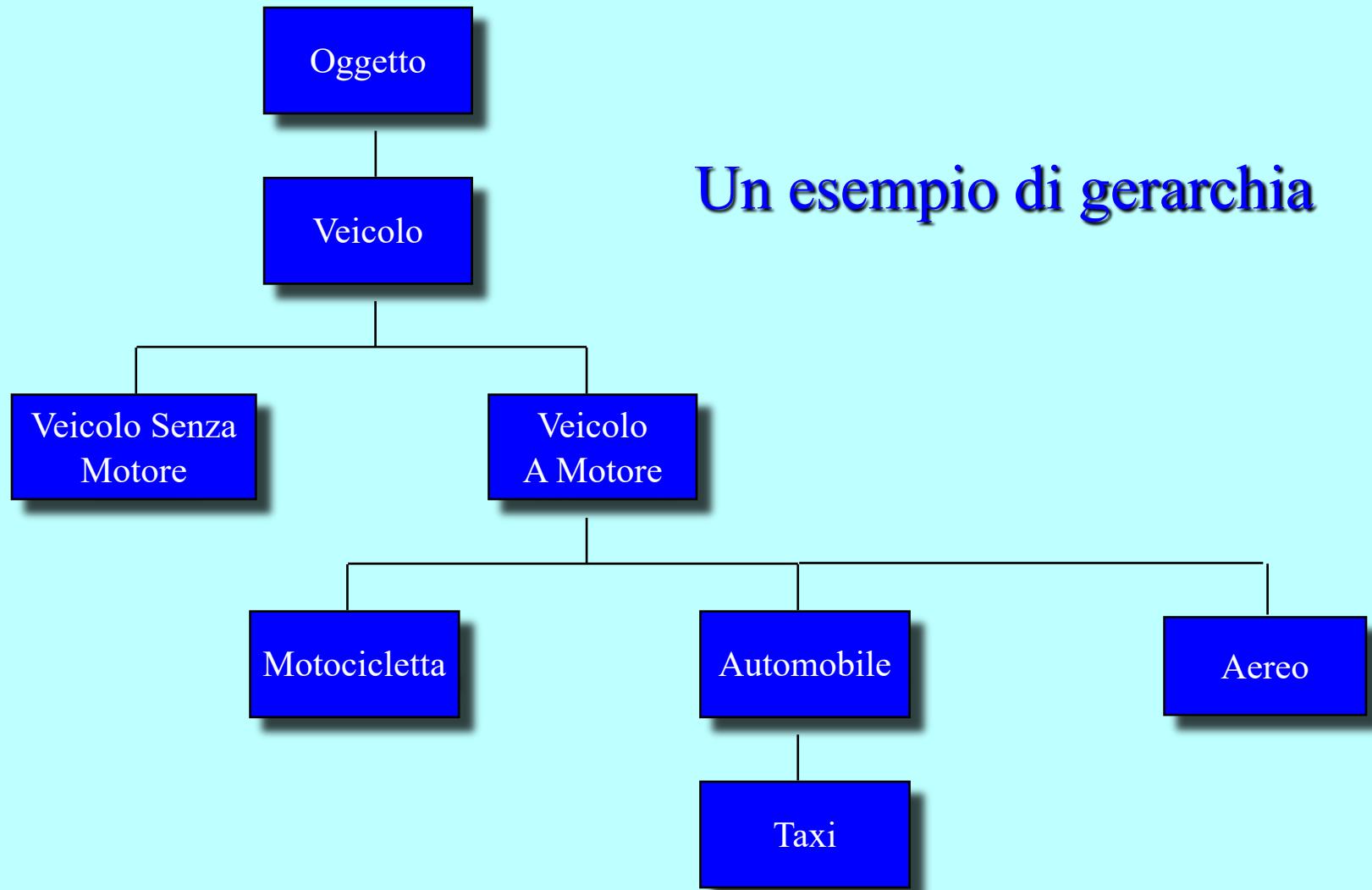
- ◆ L'ereditarietà è di fondamentale importanza nella modellazione ad oggetti, in quanto induce una strutturazione gerarchica nel sistema software in costruzione
- ◆ Essa consente di realizzare relazioni tra classi di tipo generalizzazione-specializzazione, in cui una **classe**, detta **base**, realizza 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



Ereditarietà (3/5)



Ereditarietà (4/5)

- ♦ Oltre a poter descrivere un sistema secondo un modello gerarchico, esiste un altro motivo, di ordine pratico, per cui conviene usare l'ereditarietà; esso è legato 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
- ◆ Favorisce **riuso, manutenibilità, incrementalità**

Polimorfismo (1/4)

- ◆ Per polimorfismo dinamico (o semplicemente 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:

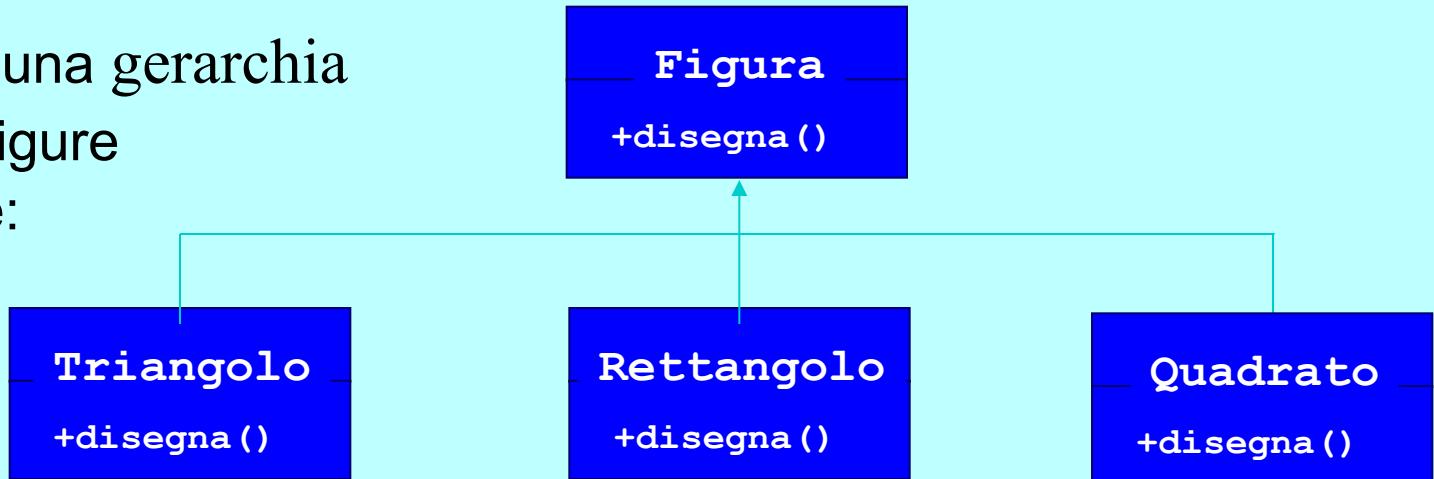
$a := b$

- ◆ L'esecuzione della operazione m sull'oggetto a dopo l'assegnazione, per la quale è tipicamente adoperata la sintassi:
$$a.m()$$
- ◆ produce l'esecuzione della implementazione di m specificata per la classe B .

Polimorfismo (2/4)

■ Esempio:

Si consideri una gerarchia di classi di figure geometriche:



- ◆ Sia ad es. A un vettore di N oggetti della classe *Figura*, composto di oggetti delle sottoclassi *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** (a tempo di 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 modificato in conseguenza dell'aggiunta di una nuova sottoclass di `Figura` (ad es.: `Cerchio`), anche se tale sottoclass non era stata neppure prevista all'atto della stesura della funzione `Disegna_Figure()`

Polimorfismo (4/4)

- ♦ Il polimorfismo supporta dunque la proprietà di **estensibilità** di un sistema, nel senso che minimizza la quantità di codice da 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 dell'orientamento ad oggetti

- ❖ **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;
- ❖ **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.

Vantaggi dell'orientamento ad oggetti

- ❖ Con l'O-O il software non viene costruito progettando separatamente i dati (con tecniche di progettazione di basi di dati) e i programmi, ma pensando a dati e operazioni in maniera integrata (classi/oggetti)
- ❖ I principi dell'orientamento agli oggetti (O-O) non riguardano solo la fase di codifica, ma possono essere adoperati anche in fase di analisi&specifica dei requisiti e in fase di progettazione
- ❖ Se dunque si effettuano l'analisi O-O, la progettazione O-O e poi la codifica O-O in un linguaggio di programmazione a oggetti come C++ o Java, **si ha il grande vantaggio di adoperare lo stesso modello (quello appunto dell'*object orientation*, che consiste dei concetti di classe, oggetto, ereditarietà e polimorfismo) in tutte e tre queste fasi del ciclo di vita del software**
- ❖ Di conseguenza, sarà più diretto il passaggio da un modello di analisi ad oggetti (ad es., con UML) ad un progetto ad oggetti (con UML), e da un progetto ad oggetti a una codifica ad oggetti