Corso di Informatica

Modulo T1

C2–Ereditarietà e polimorfismo

Prerequisiti

- Tecnica elementare di **OOP**
- Concetto di struttura gerarchica
- Concetto di *casting*

Introduzione

Abbiamo accennato al fatto che la **OOP** offre:

- un *alto livello di astrazione sui dati e procedurale*;
- un'elevata possibilità di riutilizzo del software (ridefinizione e/o espansione di classi esistenti).

Vogliamo approfondire questi concetti e in particolare rispondere alle seguenti domande:

In cosa consiste il riuso del software ?
Perché in OOP la manutenzione del software è facilitata ?

Rispondiamo introducendo alcuni concetti

Ereditarietà

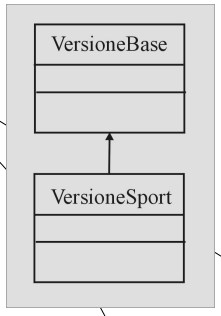
Immaginiamo un modello di vettura. Essa viene prodotta in "versione base" ed in "versione sport".

Ciò significa che il "modello sport" possiede le stesse caratteristiche del "modello base", ma in più presenta altri requisiti **specifici**.

In altra parole, il "modello sport" **deriva** dal "modello base", ossia ne **eredita** le proprietà e le funzionalità.

La programmazione OOP è caratterizzata da una proprietà detta ereditarietà, che consiste nel riutilizzare software preesistente, apportando le sole modifiche necessarie al problema in esame.

Ereditarietà



La classe *VersioneSport* si dice anche **sottoclasse** o **classe derivata** di *VersioneBase* , mentre *VersioneBase* si dice **superclasse** o **classe base** di *VersioneSport* .

La **riusabilità** del codice comporta:

- diminuzione dei **tempi di sviluppo**
- possibilità di **espansione** del codice esistente
- possibilità di **ridefinizione** del codice esistente
- **indipendenza** dal codice sorgente usato nella classe base
- **testing** del solo codice espanso

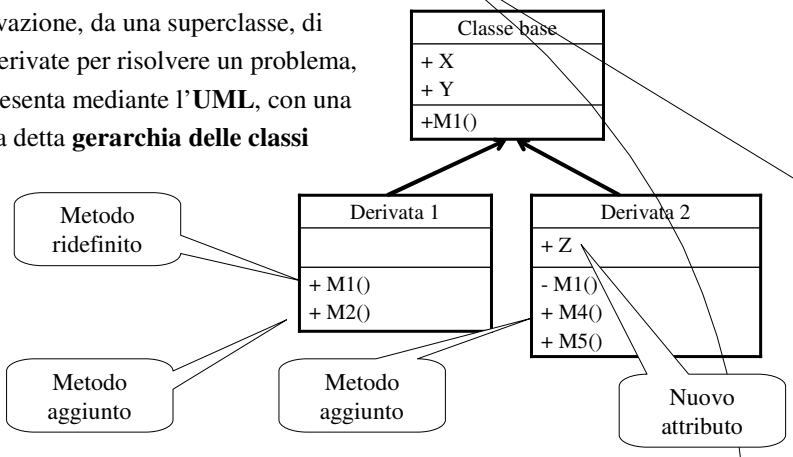
Ereditarietà

Una classe derivata, in particolare:

- **eredita** tutti e soli i **membri pubblici** della classe base
- può contenere **espansioni dei membri** della classe base (nuovi attributi e nuovi metodi) che si aggiungono a quelli della superclasse
- può richiedere la **ridefinizione** di metodi (**overriding**) ossia fornire una nuova implementazione di uno o più metodi ereditati dalla superclasse

Gerarchia delle classi

La derivazione, da una superclasse, di classi derivate per risolvere un problema, si rappresenta mediante l'**UML**, con una struttura detta **gerarchia delle classi**



Metodi di default nelle sottoclassi

La relazione di ereditarietà fa sì che anche i *metodi di default* (costruttori e distruttore) seguano un *preciso ordine di esecuzione* quando sono istanziati nelle sottoclassi.

In particolare:

- il <u>costruttore</u> di una classe derivata:
  - chiama *per prima cosa il costruttore della sua classe base* per inizializzare gli attributi e poi il proprio;
  - se è omesso, *chiama direttamente il costruttore di default della classe base*
- il <u>distruttore</u> di una classe derivata:
  - viene eseguito in ordine inverso rispetto ai costruttori: chiama *per ultimo il distruttore della sua classe base*

Ereditarietà e relazioni

L'ereditarietà può avere molte varianti e ciò offre grande flessibilità di programmazione.

L'ereditarietà dà luogo a diversi tipi di **relazioni** tra le classi.

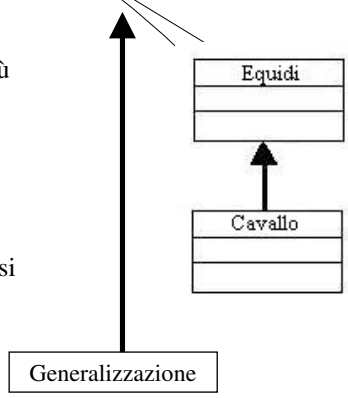
Possiamo avere re:

- Relazione di **generalizzazione / specializzazione**
- Relazione di **aggregazione / composizione**

Generalizzazione e specializzazione

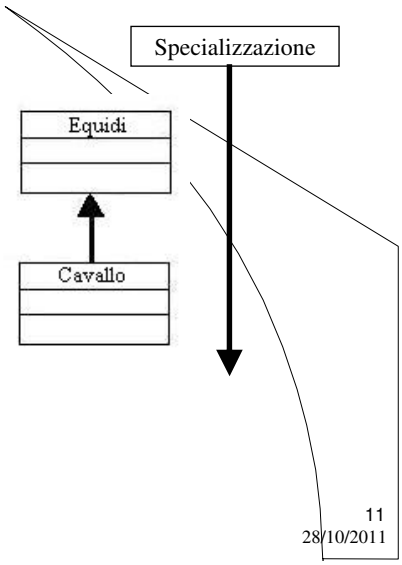
La **generalizzazione**, indica che la sottoclasse deriva da una classe più generale.

L'esempio, si legge dicendo che "Un cavallo **è un** equide", avendone ereditato le caratteristiche. Questa relazione di ereditarietà, pertanto, si dice "**is a".**



Generalizzazione e specializzazione

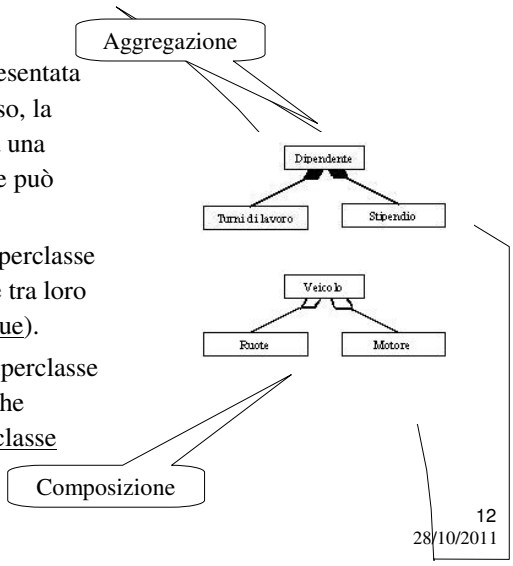
La **specializzazione,** indica che la sottoclasse ha rispetto alla superclasse delle caratteristiche particolari, più specifiche



Aggregazione e composizione

Una superclasse può essere rappresentata da più sottoclassi. In questo caso, la gerarchia delle classi individua una **relazione di contenimento** che può distinguersi in:

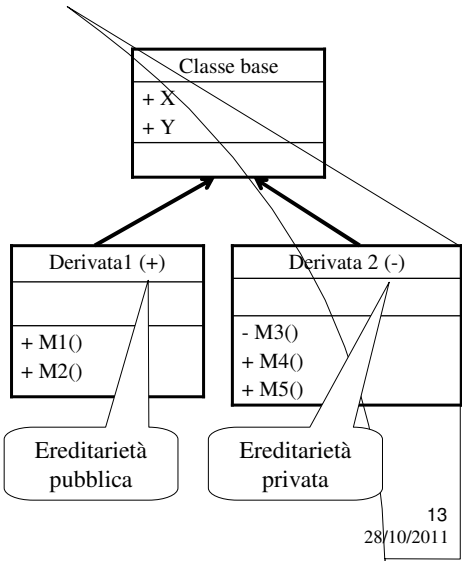
- **aggregazione,** indica che la superclasse è formata da più classi derivate tra loro indipendenti (esistono comunque).
- **composizione,** indica che la superclasse è composta da più sottoclassi che esistono solo se esiste la superclasse



Ereditarietà pubblica o privata

Una classe derivata può modificare la **visibilità dei membri** della classe base, dichiarando tutto ciò che eredita dalla classe base come:

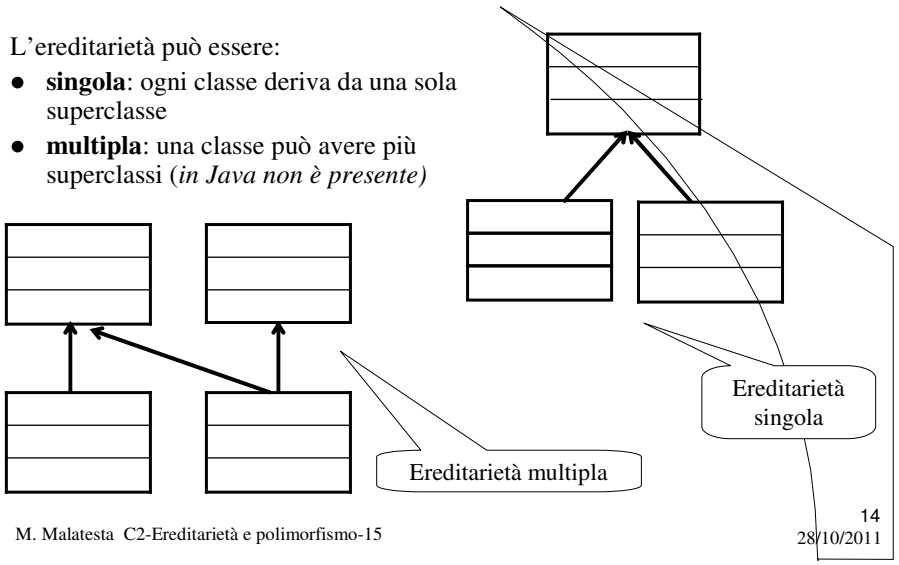
- **pubblico:** ogni membro pubblico della superclasse, resta pubblico anche nella sottoclasse (**ereditarietà pubblica)**
- **privato:** ogni membro pubblico della superclasse, diventa privato nella sottoclasse (**ereditarietà privata**)



Ereditarietà singola o multipla

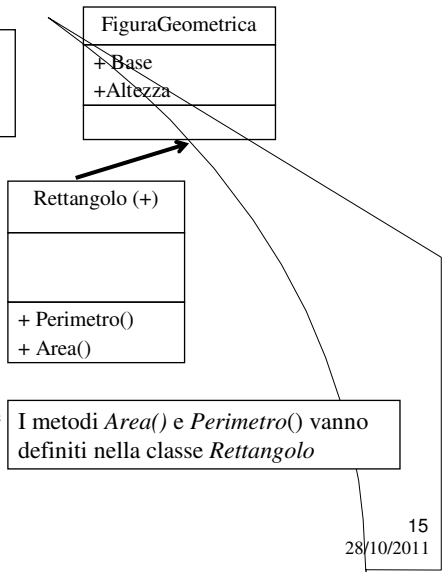
L'ereditarietà può essere:

- **singola**: ogni classe deriva da una sola superclasse
- **multipla**: una classe può avere più superclassi (*in Java non è presente)*



Espansione e ridefinizione

ATTIVITA': Dalla classe *FiguraGeometrica* derivare la classe *Rettangolo* con ereditarietà pubblica



La sottoclasse *Rettangolo*

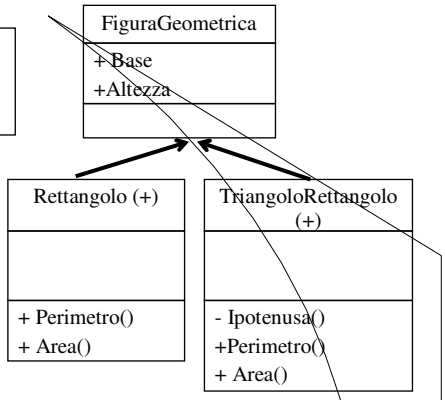
- **deriva** dalla superclasse *FiguraGeometrica* per **eredità pubblica**
- **eredita tutti i membri** ed **espande** la classe base *FiguraGeometrica* con i metodi *Perimetro()* e *Area()*.

Un oggetto di classe *Rettangolo* può utilizzare entrambi i metodi *Perimetro()* e *Area()* ed entrambi gli attributi *Base* e *Altezza*.

I metodi *Area()* e *Perimetro*() vanno definiti nella classe *Rettangolo*

Espansione e ridefinizione

Dalla classe *FiguraGeometrica* derivare la classe *TriangoloRettangolo* con ereditarietà pubblica



La classe derivata *TriangoloRettangolo*

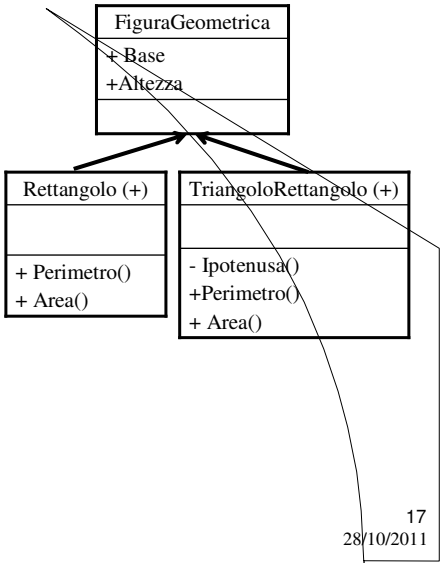
- **deriva** dalla superclasse *FiguraGeometrica* per **eredità pubblica**
- **eredita tutti i membri** ed **espande** la classe base *FiguraGeometrica* con i metodi *Perimetro()* e *Ipotenusa()* (*privato*, quindi non utilizzabile all'esterno della classe);

I metodi *Area()* e *Perimetro*() vanno **ridefiniti** nellae classe *TriangoloRettangolo*, poiché calcolati in modo diverso rispetto alla classe *Rettangolo*

Espansione e ridefinizione

Un oggetto di classe *TriangoloRettangolo* può utilizzare entrambi i metodi *Perimetro()* e *Area()* ed entrambi gli attributi *Base* e *Altezza*

Le classi derivate hanno **eredità pubblica** (caratterizzata dal segno **"+"** a fianco) e quindi mantengono pubblico ciò che hanno ereditato. In caso contrario, il segno **"-"** indicherebbe che ciò che ereditano diventa privato.



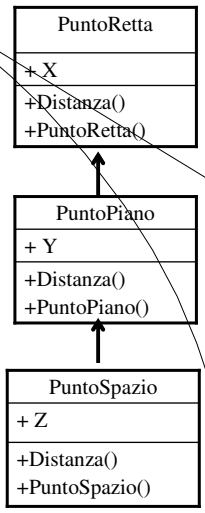
Espansione e ridefinizione

Nella costruzione della gerarchia delle classi occorre procedere alla **generalizzazione** e creare dapprima *una superclasse più generale possibile*. In questo modo la creazione delle classi derivate presenta i vantaggi dell'ereditarietà che abbiamo elencato

Dalla classe *PuntoRetta* derivare la classe *PuntoPiano* con ereditarietà pubblica

Dalla classe *PuntoPiano* derivare la classe *PuntoSpazio* con ereditarietà pubblica

Nella derivazione è necessario **espandere** una seconda coordinata e **ridefinire** i metodi *Distanza()* e il costruttore



Ereditarietà e indipendenza

L'ereditarietà è utile poiché:

- **classi diverse sono spesso molto simili**.
  In questi casi si racchiudono nella classe base i *membri comuni* e si usa l'ereditarietà per costruire le classi derivate con i *soli membri specifici* di queste
- **le classi sono indipendenti**
  Eventuali modifiche nella classe base, *non coinvolgono modifiche nelle classi derivate* (ovviamente finché le interfacce, cioè attributi e metodi **public** rimangono invariati). Il **collaudo** o **testing** può essere *limitato alla sola parte nuova di codice aggiunto nelle sottoclassi*. Le classi derivate potrebbero dover essere ricompilate

Visibilità dei membri

Abbiamo visto che gli specificatori di accesso (**public** e **private**) limitano l'accesso ai membri di una classe.

Se una classe ha un membro private come si può accedere ad esso da parte di una classe derivata?

Visibilità di un membro-protected

Abbiamo le seguenti possibilità:

- definire **public** il membro, ma in tal caso si presenta il rischio di un utilizzo improprio da parte del programma (violazione dell'*information hiding*)
- lasciare **private** il membro, ma fornire nella sua classe un metodo per il suo utilizzo
- una alternativa è l'utilizzo di un terzo specificatore di accesso detto **protected.** In tal caso esso è accessibile soltanto ai metodi interni alla classe derivata, ma ancora è presente il rischio di violazione dell'*information hiding*. Per evitare ciò, spetta alla classe derivata definire un metodo per accedere al membro protetto. In **UML** il membro **protected** si indica con il simbolo **"#"**

Conversione

Classi base e derivate *sono di tipo differente*, per cui quando si esegue un'assegnazione tra esse occorre prestare attenzione.

In questi caso, si parla di **conversione di classe** (*casting*).

Abbiamo:

- **upcasting**, quando un oggatto di una classe derivata viene convertito in uno di una superclasse
- **downcasting**, quando un oggetto di una classe viene convertito in uno di una sottoclasse.

Upcasting

Un oggetto di una classe derivata PUO' essere convertito in un oggetto della classe base. Infatti

- la classe derivata ha membri corrispondenti a tutti i membri della classe base (tranne i privati)
- la classe derivata ha di solito più membri della classe base e tutt'al più possiamo perdere dei membri

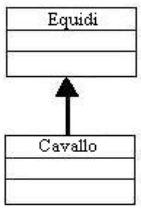
Equide eq = **new** Equide();

Cavallo c = **new** Cavallo();

è possibile convertire un oggetto di classe *Cavallo* in uno della sua superclasse:

eq = c;

Upcasting



Downcasting

Un oggetto di classe base NON può essere convertito in un oggetto di classe derivata. Infatti scrivendo

c = eq; oppure c = (Cavallo) eq;

- *c* avrebbe dei dati membro indefiniti
- il tipo di arrivo (*Cavallo*) è più ampio di quello di partenza (*Equide*)

Il downcasting è ammesso quando l'oggetto origine sia della stessa classe di quello di destinazione, ossia:

Equide eq = **new** Cavallo(); /* casting automatico */

Cavallo c = (Cavallo) eq; /* casting esplicito */.

Downcasting

Polimorfismo

Il **polimorfismo** può presentarsi in due modalità differenti:

- **overloading**
- **overriding**

Polimorfismo
Overloading

Consideriamo alcuni esempi per descrivere il polimorfismo.

1) Si considerino I seguenti esempi di utilizzo del metodo **println**() sull'oggetto **System.out**:

- **System.out.println** ("Ciao");
- **System.out.println** (2*a-b); (con a e b **int**)
- **System.out.println** (**Math**.**PI** * **Math.pow**(r,2));

Sullo *stesso oggetto* **System.out** attiva il metodo di stampa **println**() in forme diverse. Infatti, è in grado di stampare:

- una **stringa** costante;
- un'espressione con variabili **int**;
- un'espressione con variabili **double**.

Polimorfismo
Overloading

2) un **doppio click del mouse**, può avere diversi significati: aprire una cartella, eseguire un programma, selezionare un testo.
3) nella classe *Punto* possiamo implementare due metodi *PMedio*() (uno statico e uno non statico) aventi lo stesso nome, *ma firma diversa*.
4) nella stessa classe *Autovettura*, il metodo *Accelera*() può essere:
   - senza parametri, e in questo caso ad esempio, rappresenta una accelerazione prestabilita costante;
   - con parametri, come Accelera (**int** v), per trattare il caso in cui è richiesto portarsi ad una velocità v
5) una classe può contenere più costruttori con firma diversa (senza parametri oppure con parametri)

Questi descritti sono esempi in cui si hanno **metodi in sovraccarico** (**overloading**)

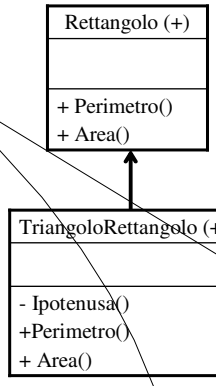
Polimorfismo
Overloading

Il **sovraccarico** (*overloading*) di metodi implica che una stessa azione (metodo) possa presentarsi in diverse forme al momento dell'esecuzione.

Diversamente dalla programmazione procedurale, in cui ciò non è ammesso, in **OOP** questa eventualità è ammessa e come sappiamo è una proprietà detta **polimorfismo**.

Con il termine **sovraccarico dei metodi** (*overloading*) intendiamo la possibilità di avere più metodi con lo stesso nome all'interno di una **medesima classe,** che **differiscano per la firma**.

Polimorfismo
Overriding



Se *Rettangolo* è superclasse di *TriangoloRettangolo,* i metodi ereditati *Area*() e *Perimetro*() devono essere ovviamente riscritti nella classi *TriangoloRettangolo*, poiché è diverso il procedimento di calcolo.

In questo caso, il metodo ereditato nasconde quello della superclasse: questa situazione viene detta **oscuramento dei metodi** (*overriding*) e ovviamente richiede la riscrittura del codice del *metodo ereditato da una sottoclasse.*

Con il termine **oscuramento dei metodi** (*overriding*) intendiamo la presenza di più **metodi con lo stesso nome** ma **firma diversa**, **all'interno di una gerarchia di classi**.

Utilizzo dell'ereditarietà

Nella **Fase 1** (**OOA**), durante la progettazione delle classi relative al problema, occorre:

- **generalizzare le classi**, mediante individuazione delle proprietà comuni, e costruzione della **classe base** sfruttando l'**incapsulamento**
- **specializzare la classe base** in classi derivate, mediante l'**ereditarietà**
- **specializzazione eventuale dei metodi** mediante il **polimorfismo** (*overloading e/o overriding* dei metodi)

Metodi e classi astratte

Nella creazione di una gerarchia di classi che risolve un problema, la superclasse può essere *composta da metodi non implementati*, dei quali compare la sola interfaccia, in modo da renderla *più generale possibile*.

Questi metodi si dicono **metodi astratti** e sono caratterizzati dalla clausola **abstract**.

Una classe che contenga *almeno un metodo astratto* si dice **classe astratta** ed è caratterizzata dalla clausola **abstract**

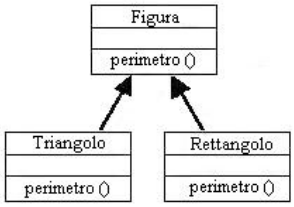
Metodi e classi astratte

Possiamo osservare che una classe astratta:

- *non può essere istanziata*, in quanto un eventuale oggetto creato mancherebbe dei comportamenti relativi ai metodi non implementati;
- di solito è utilizzata per contenere un *elenco dei membri da ridefinire* poi in ciascuna sottoclasse della gerarchia;
- non può essere dichiarata come *classe finale* di una gerarchia di classi.

Metodi e classi astratte

Ad esempio, nella gerarchia *FigureGeometriche* la classe *Figura* è una classe astratta, poiché, non essendo noto il numero dei lati, il metodo *perimetro* () non può essere implementato; esso deve esser ridefinito opportunamente in ciascuna delle classi derivate *Triangolo* e *Rettangolo*.



L'ereditarietà si rivela uno strumento di grande potenza, poiché attraverso la creazione di classi astratte, è possibile un alto grado di riuso del software, in particolare per le classi grafiche di cui Java possiede molte varianti.

Argomenti

- Ereditarietà
- Gerarchia delle classi
- Metodi di default nelle sottoclassi
- Ereditarietà e relazioni
- Generalizzazione e specializzazione
- Aggregazione e composizione
- Ereditarietà pubblica o privata
- Ereditarietà singola o multipla
- Espansione e ridefinizione
- Ereditarietà e indipendenza
- Visibilità dei membri
- Visibilità di un membro-protected
- Conversione
- Upcasting
- Downcasting
- Polimorfismo
  - Overloading
  - Overriding
- Utilizzo dell'ereditarietà
- Metodi e classi astratte

Altre fonti di informazione

- P.Gallo, F.Salerno – Informatica Generale 1, ed. Minerva Italica
- M.Romagnoli, P.Ventura – Linguaggio C/C++, ed. Petrini
- A. Garavaglia, F.Petracchi, S.Forte
  Strutture dati e programmazione per oggetti, ed. Masson Scuola