INTRODUZIONE ALLA OOP (OBJECT ORIENTED PROGRAMMING)

Perché serve?

Il paradigma di programmazione che abbiamo utilizzato finora con il linguaggio C è chiamato imperativo procedurale. Ora introduciamo un nuovo paradigma di programmazione chiamato Object Oriented Programming (OOP).

Il grande vantaggio della OOP, rispetto alla programmazione imperativa procedurale, è quello di facilitare il riutilizzo del codice già scritto.

Prendiamo l'esempio del progetto "Gestione Studenti". Immaginiamo questo progetto come come se fosse una parte di una applicazione del tipo "registro elettronico" realizzata per una scuola che chiamiamo scuola A. Supponiamo di aggiungere alcune funzioni, ad esempio la funzione "modificaIndirizzo" che consente di modificare l'indirizzo di uno studente e la funzione calcolaEta che consente di calcolare l'età di uno studente.

Supponiamo poi che un'altra scuola (chiamiamola scuola B) ci chieda di sviluppare un software per la gestione anagrafica degli studenti da utilizzare in segreteria. Questa nuova applicazione dovrà utilizzare sempre la struttura "struct tipoStudente". Visto che abbiamo già tale struct ed alcune operazioni su di essa, sarebbe molto vantaggioso riutilizzare il codice già scritto. Per riutilizzare il codice dovremmo copiare/incollare tutto ciò che riguarda il tipo di dato tipoStudente dal codice della prima applicazione (registro elettronico della scuola A) nel codice della seconda applicazione (anagrafica studenti della scuola B).

Dovremmo andare a copiare/incollare nel nuovo codice, la definizione della struttura tipoStudent e tutte le funzioni che riguardano lo studente (la funzione che calcola l'età, la funzione che modifica i l'indirizzo ecc..). Operando in questo modo si rischierebbe di dimenticarsi di copiare alcune funzioni. Inoltre le funzioni che andremmo a copiare, per poter essere riutilizzate, potrebbero aver bisogno di modificare alcuni parametri che nel nuovo programma non ci sono, perché il nuovo programma fa altro.

Per questi motivi riutilizzare il codice, soprattutto se il progetto è complesso, diviene molto difficile.

Come sarebbe bello se noi disponessimo di un file che contiene tutto ciò che riguarda il tipo di dato "studente" (le sue caratteristiche come il cognome, la matricola ecc, e anche tutte le funzioni che è possibile svolgere su di esso, tipo calcolarne l'eta, modificarne i dati ecc..). Per riutilizzare il tipo di dato studente basterebbe, in questo caso, importare il file "studente" nel nuovo progetto, e tutto funzionerebbe, senza dover modificare funzioni ecc…

Questo è ciò che accade con la programmazione ad oggetti **OOP** (**object oriented programming**).

Certo, anche nella programmazione tradizionale che abbiamo imparato con il linguaggio C, che si chiama imperativa-procedurale, se la struttura "tipoStudente" e tutte le funzioni che agiscono sul tipo di dato "tipoStudente", fossero scritte in maniera ordinata in unico file (ad esempio "studente.c"), allora sì, sarebbe comodo e veloce importare il file in un nuovo progetto ed avere a disposizione tutto il codice relativo al tipo di dato "tipoStudente". Il problema è che nel caso della programmazione procedurale questo *si potrebbe* fare, ma non c'è l'obbligo. E' a discrezione del programmatore raggruppare una struttura dati con tutte le sue funzioni in un file, ma il programma funziona anche se ciò non avviene. Ma allora, quando un altro programmatore volesse utilizzare il file "studente.c" potrebbe chiedersi: "ma siamo sicuri che c'è dentro tutto?", magari una funzione <<modificaIndirizzo>> potrebbe essere stata scritta nel Main.c anziché nel file studente.c semplicemente perché al programmatore di "studente.c" risultava comodo così, o perché in quel momento gli pareva giusto così. Quindi con la programmazione procedurale è possibile **ma non obbligatorio** mantenere insieme le strutture dati e le funzioni che agiscono su di esse.

Con la **programmazione ad oggetti** invece questo diviene **obbligatorio**. Ossia i tipi di dato sono costruiti obbligatoriamente in modo che ci sia sempre un contenitore, una "capsula", che contiene "tutto" di quel tipo di dato, ossia la struttura (cognome, nome, matricola, ecc) e le funzioni che permettono di agire su di esso (calcolaEta, cambiaIndirizzo, ecc.). Se il tipo di dato non è incapsulato, il programma non funziona, quindi il programmatore è OBBLIGATO a costruire tipi di dato in cui tutto ciò che riguarda quel tipo di dato è racchiuso nella "capsula". Quindi chi riutilizzerà quel tipo di dato sarà CERTO che tutte le funzioni relative a quel tipo di dato saranno nella capsula.

I tipi di dato astratti (ADT)

I tipi di dato che conosciamo (int, double, char ecc…) sono sempre formati da due componenti: il contenuto informativo e un insieme di operazioni che possono essere svolte su di essi. Il **contenuto informativo** dice "quanto valgono" (il valore che assumeranno poi le variabili definite con tali tipi di dato –attenzione, sottolineo che stiamo sempre parlando di tipi di dato, non di variabili -). L'insieme di operazioni che possono essere svolti su di essi sono, ad esempio: sommare, sottrarre, calcolare la radice quadrata, concatenare (nel caso delle stringhe) confrontare…. Nella programmazione procedurale queste due componenti, componente informativa e operazioni, sono separate (infatti si possono definire le strutture dati in un punto del codice e le funzioni anche in un altro punto o un altro file). Nella programmazione OOP la componente informativa e la componente "funzionale" dei tipi di dato sono invece unite. Questi tipi di dato, definiti nella OOP, sono chiamati **classi** e fanno parte dei cosiddetti **tipi di dato astratti**.

Un tipo di dato astratto (**abstract data type, ADT**) è un tipo di dato costituito da:

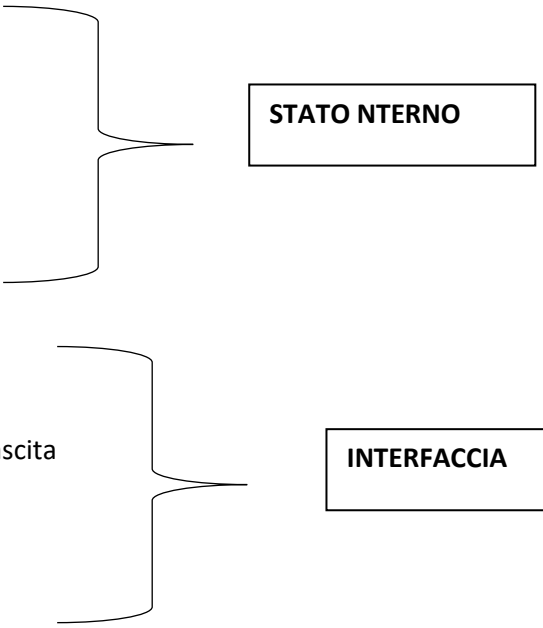
- una componente informativa che definisce lo **stato interno** del tipo di dato. Noi chiameremo questa componente informativa: **attributi**
- una serie di operazioni per agire sullo stato interno del tipo di dato, ossia per agire sugli attributi. Chiameremo queste operazioni: **metodi**

La caratteristica del ADT è quella di consentire la modifica dei suoi attributi (ossia del suo stato interno) da parte del programmatore solamente attraverso i metodi che il tipo di dato stesso mette a disposizione, e che sono indicati con il termine **interfaccia**.

Esempio:

ADT Studente

- **Attributi**:
  - Cognome
  - nome
  - indirizzo
  - citta
  - provincia
  - giorno
  - mese
  - anno
- **Metodi** (funzioni):
  - assegnaNome
  - …..
  - assegnaGiornoNascita
  - ……
  - calcolaEta
  - …..



Quando un programmatore dichiarerà una variabile (che nella OOP chiameremo **oggetto**) di tipo "Studente", non potrà, ad esempio, assegnare direttamente un valore all'attributo "giorno" della data di nascita, ma dovrà utilizzare il metodo assegnaGiornoNascita. I metodo sono simi alle funzioni viste in C.

studente1.giorno=23; Non si può assegnare direttamente

studente1.assegnaGiornoNascita(23); Si assegna invocando l'apposito metodo.

Riassumendo, dunque, cosa è un ADT?

E' un tipo di dato formato sia dalla componente informativa (ATTRIBUTI) sia dalle operazioni che il programmatore può svolgere su di esso (METODI), in cui l'insieme dei metodi è chiamato INTERFACCIA, e che ha nell'INTERFACCIA l' UNICO MECCANISMO tramite il quale è possibile interagire con il dato, ossia modificare o rilevare lo stato interno. Lo stato interno è dato dal valore degli attributi.

Information Hiding

Il principio che sta alla base della progettazione degli ADT è quello dell' "**information hiding"** ossia dell' occultamento dell'informazione.

Con questo concetto si indicano due cose:

1. il fatto che quando un programmatore **utilizza** un tipo di dato astratto, dovrà accedervi solamente tramite l'interfaccia, quindi non è necessario che egli conosca il codice interno con cui è stato realizzato l' ADT. Tale codice rimarrà quindi "nascosto", "hiding".
2. Quando un programmatore **utilizza** un tipo di dato astratto egli non può accedere direttamente allo stato interno del tipo di dato, ma può farlo solo attraverso l'interfaccia.

Il principio dell'information hiding viene implementato nella OOP con la tecnica **dell'incapsulamento**. Questa tecnica consiste nel "proteggere" lo stato interno di un tipo di dato per mezzo dell'interfaccia, nel renderlo inaccessibile, tranne che con l'apposita interfaccia, come se tale stato interno fosse racchiuso in una capsula di protezione a cui il programmatore che utilizza l' ADT può accedere solo attraverso l'interfaccia.

Quale è il vantaggio di consentire l'interazione con gli ADT solamente attraverso l'interfaccia?

I vantaggi sono due:

1) i metodi dell'interfaccia possono contenere tutti i controlli che consentono un utilizzo corretto dell' ADT. Per esempio: se il programmatore volesse creare una variabile di tipo "studente" ed assegnare all'attributo *giorno* il valore 23, dovrà usare il metodo "assegnaGiornoNascita(23)". Questo perché il metodo "assegnaGiornoNascita (int giorno)" è come una "funzione", e dentro quella funzione ci potranno essere tutti i controlli che rendono possibile assegnare all'attributo "giorno", un valore compreso fra 1 e 31. La funzione che controlla che il valore assegnato sia corretto, non sarà "in giro" da qualche parte nel codice. Grazie a questo, se il programmatore volesse usare "studente" in un altro programma, non avrà il problema di dover copiare la parte di codice di controllo sull'attributo "giorno", e magari di dimenticarsi di copiare tale controllo, perché tale controllo è "dentro" l'ADT "studente", fa parte dell'ADT, è già incapsulato nell' ADT.
2) Un' eventuale modifica del codice che **implementa** l'ADT, ad esempio la modifica del codice di un metodo (refactoring) per migliorarne l'efficienza, oppure la correzione di un errore in uno dei metodi dell' ADT, può essere apportata senza che sia necessario modificare il codice che **utilizza** l'ADT. Infatti se venisse modificato il codice di un metodo, lasciando inalterata l'interfaccia, il programma che utilizzava l'ADT prima della modifica potrà utilizzare l'ADT modificato senza bisogno di modificare alcuna riga di codice.

L'incapsulamento è la tecnica con la quale si implementa il concetto teorico dell'**information hiding**. Quest'ultimo è uno dei tre pilastri su cui fonda la OOP, gli altri sono: ereditarietà e polimorfismo (che vedremo in seguito).

Attenzione: non tutti i metodi dell'ADT fanno parte dell'interfaccia. O meglio, l'interfaccia è costituita dai soli metodi che il progettista dell' ADT vuole mettere a diposizione del programmatore per interagire con gli attributi, tali metodi sono detti **pubblici**, esistono però metodi che vengono usati internamente alla struttura dati e che quindi non sono resi accessibili al programmatore che utilizza l'ADT, questi ultimi sono detti metodi **privati**.

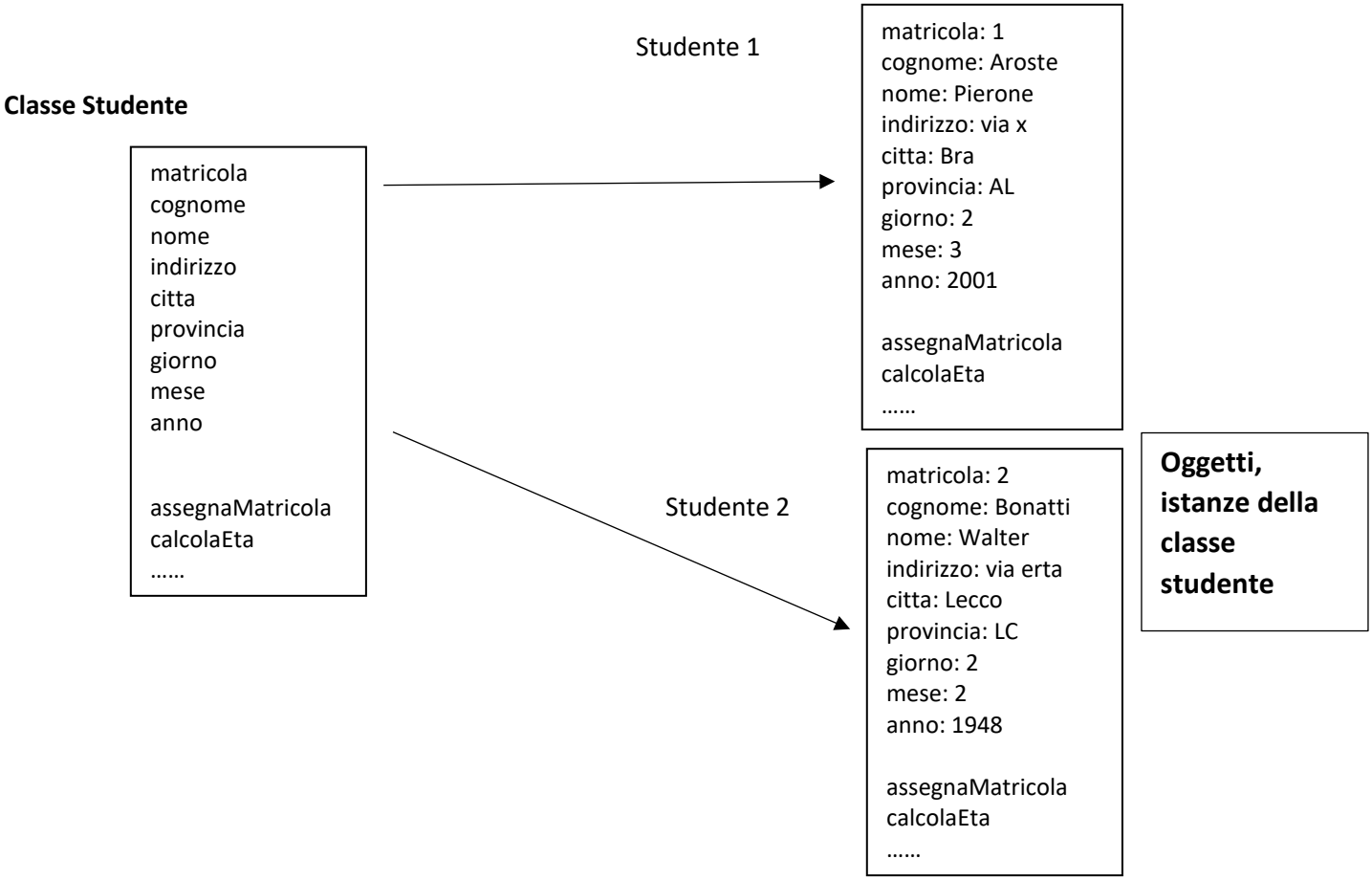
CLASSI E OGGETTI

Gli ADT nella OOP vengono implementati realizzando degli strumenti sw chiamati **classi**. Le classi sono dei **modelli formali** di oggetti, esse definiscono quali sono e come sono gli **attributi** e i **metodi** di quel tipo di oggetto.

Dalle classi derivano, o meglio, "vengono **istanziati**", degli **esemplari** chiamati **oggetti**.

Ad esempio "studente" è una classe, in essa sono specificati tutti gli attributi e i metodi che deve avere una oggetto studente (matricola, cognome, ecc..).

Dalla classe "studente" vengono istanziati gli **oggetti** studenti nel momento in cui si creano degli elementi sw ti tipo studente ai quali vengono assegnati dei valori. L'interazione fra oggetti diversi avviene attraverso lo scambio di **messaggi**. Si ha uno scambio di messaggio quando un oggetto invoca un metodo di un altro oggetto.



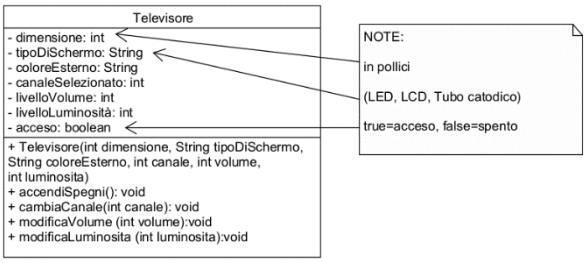
OSSERVAZIONE INTERESSANTE

La programmazione ad oggetti è un'evoluzione della programmazione procedurale imperativa, si basa su di essa, non è un cambiamento ma un passo in avanti, infatti le strutture di controllo della programmazione procedurale (sequenza, selezione, iterazione) vengono utilizzate anche nella OOP. La OOP consente di creare moduli riutilizzabili (le classi) possono essere utilizzate a loro volta come componenti per creare altre classi (studente si può usare per realizzare la classe "ConsiglioDiClasse" costituita dalla classe studente, docente, genitore, ecc…). Questa estrema modularità deriva da un modello progettuale utilizzato da molto tempo nei processi industriali. Quando si vuole produrre industrialmente un prodotto, esso si ottiene assemblando altri prodotti già pronti, senza sapere come sono stati costruiti ma semplicemente utilizzandoli. Ad esempio quando si produce una lavatrice, si assemblano oggetti già pronti (un motore, dei pulsanti, un oblò, una cesta ecc..) ma per poter essere utilizzati, questi componenti devono rispettare degli standard (di dimensione, di materiale, ecc). La stessa cosa accade per le classi della OOP. Ogni classe è un elemento che può essere testato individualmente, una volta che si è certi che funziona bene non è più necessario sapere come è stata realizzata, un programmatore la utilizzerà come componente già pronto per realizzare altre classi e applicazioni, semplicemente interagendo con l'interfaccia che essa espone.

Esempio: classe Televisore.

Attributi (descrivono le caratteristiche dell'oggetto, in un determinato istante):

Dimensione (in pollici)
TipoDiSchermo (LED, LCD, Tubo catodico)
ColoreEsterno
CanaleSelezionato
LivelloVolume
LivelloLuminosità
AccesoSpento



Operazioni (metodi):

Televisore(int dimensione, String tipoDiSchermo, String coloreEsterno, int canale, int volume, int luminosita)

Metodo costruttore

accendiSpegni
cambiaCanale
codificaVolume
modificaLuminosità

Dalla Televisore classe si possono istanziare 2 oggetti, ad esempio l'oggetto tvSoggiorno, in cui l'attributo *Dimensione in pollici* assume valore 32, e l'oggetto tvCucina, in cui *Dimensione In pollici* vale 24. I due oggetti sono due istanze della classe televisore.

Cosa succede quando viene istanziato un oggetto:

1. Viene allocata una zona di memoria centrale in cui viene memorizzato l'oggetto.
2. Vengono inizializzati gli attributi dell'oggetto.

Questa operazione di creazione degli oggetti, chiamata **istanziazione** è svolta da un particolare **metodo** chiamato **costruttore**, che **ha lo stesso nome della classe**. Nel nostro esempio il costruttore della classe Televisore, avrà nome Televisore.

Importanza del costruttore: il costruttore è importante perché nel codice di questo metodo viene specificato cosa deve essere fatto ogni volta che viene istanziato un oggetto, compreso il codice di controllo sui valori assegnati agli attributi. Se voglio creare una nuovo oggetto "studente" e specificare che l'attributo giorno

dovrà essere un numero da 1 a 31, nel metodo costruttore, imporrò il vincolo che l'attributo "giorno" sia compreso fra 1 e 31, e ogni studente non potrà essere creato se non viene assegnato un numero corretto all'attributo giorno. Il costruttore estende il significato di inizializzazione che abbiamo visto in C.
Il costruttore ha lo stesso nome della classe e non ha valore di ritorno, neppure void.

Esempio pratico: classe Televisore, creiamola in java per capire a cosa serve il costruttore.

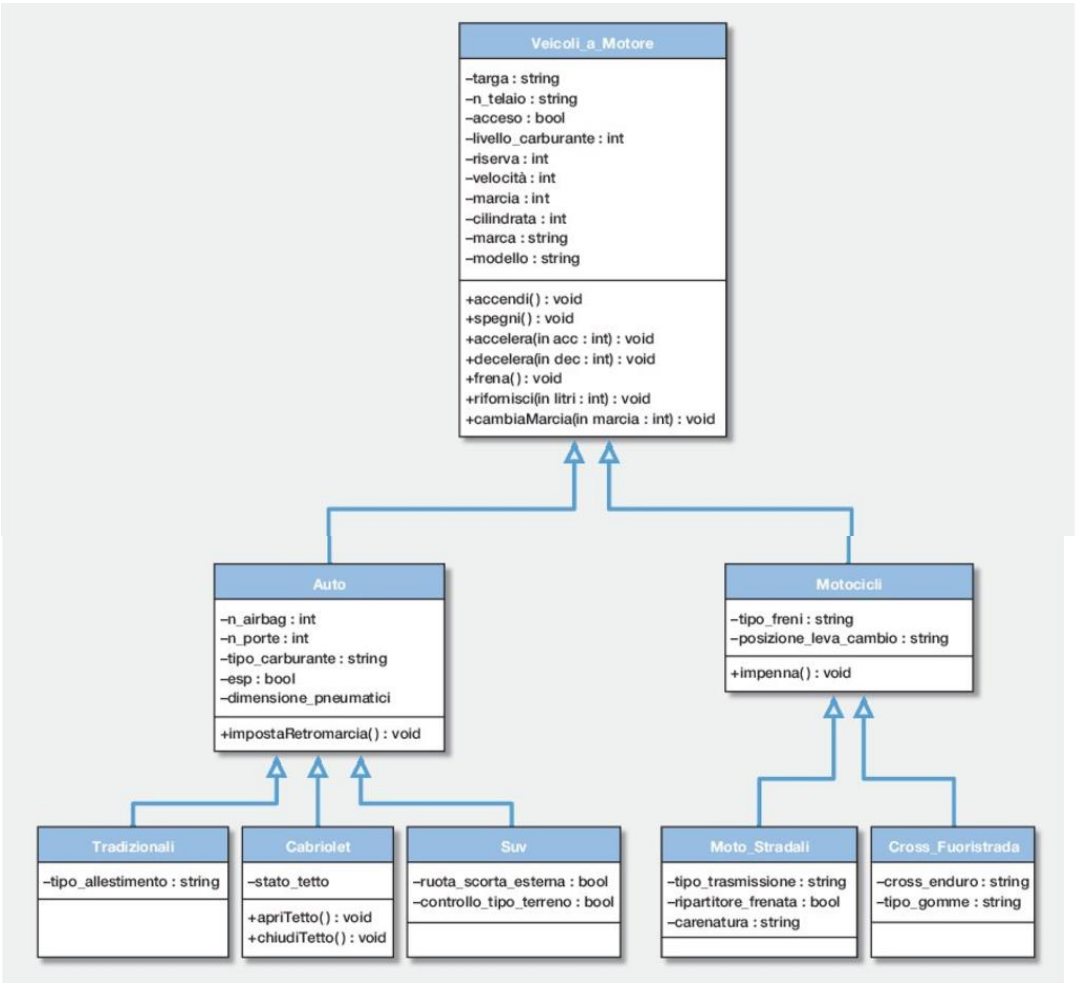
Ereditarieta'

Il secondo concetto fondamentale della OOP è l'EREDITARIERTA', con questo termine si indica un meccanismo che consente di costruire delle nuove classi a partire da classi già esistenti. Le nuove classi ereditano da quelle da cui derivano, gli attributi e i metodi, ma quando vengono definite, ad esse si possono aggiungere ulteriori metodi e attributi, inoltre si può ridefinire il codice dei metodi esistenti modificandone il comportamento.

Le classi da cui derivano altre classi sono dette **superclassi** o **classi madre** o **classi base**. Le classi **derivate** sono dette **sottoclassi** o **classi figlie**. Ogni classe figlia può a sua volta diventare una classe madre per un'altra classe realizzando così una **gerarchia** di classi.

Un esempio di gerarchia di classi con classe madre e classi figlie è il seguente. Le classi sono rappresentate con il diagramma delle classi che è un formalismo del linguaggio UML che studieremo in TPS, ma è comunque, a questo livello, comprensibile. Il diagramma delle classi mostra le classi rappresentandone il nome, gli attributi e i metodi.

Esempio p.14 del libro (Classe Veicolo_a_motore)



I metodi e gli attributi della classe “veicolo_a_motore” fanno parte di entrambe le sottoclassi Auto e Motocicli (si dice che vengono “ereditati”), infatti entrambe le sottoclassi descrivono oggetti che hanno una targa, un telaio ecc.. . Le sottoclassi, oltre ai metodi e attributi ereditati, ne possiedono altri, diversi, ciascuno specifico per le specifica sottoclasse. Lo stesso meccanismo di ereditarietà avviene per le classi figlie delle classi figlie (Tradizionali, Cabriolet, Suv, Moto_Stradali, Cross_fuoristrada).

Il vantaggio dell’ereditarietà è quello di **consentire la programmazione per differenze**, favorendo notevolmente il riutilizzo del codice.

Infatti se, ad nell’esempio precedente si volesse creare la classe dei motoscafi, essa potrebbe essere costruita come sottoclasse dalla classe dei veicoli a motore. Il vantaggio risulta evidente pensando che tutti gli attributi e i metodi dei veicoli a motore sono già disponibili senza dover riscrivere alcuna riga di codice, testati e sicuri con tutti i controlli del caso (es. *livelloCarburante* non può assumere un valore <0, *marcia* non può assumere valore<0, il metodo *cambiaMarcia* è già una “funzione” pronta per modificare il valore dell’attributo marcia ecc..)

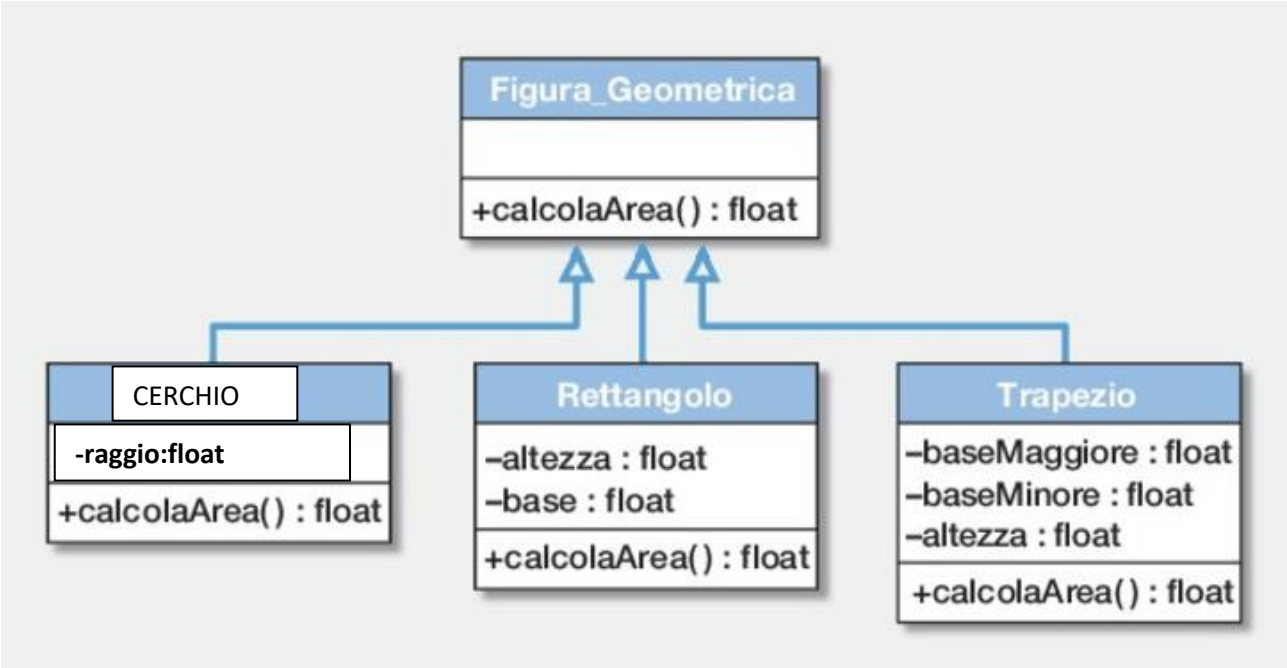
La una classe figlia è detta anche classe **specializzata** rispetto ad una classe madre. Il nome deriva dal fatto che la classe figlia può “fare tutto” ciò che può fare la classe madre **e in più può “fare altre cose” specifiche, speciali, quindi è una classe specializzata.** La classe madre è invece detta classe **generalizzata** rispetto alla classe figlia.

Polimorfismo

La terza caratteristica fondamentale della OOP è il polimorfismo, che è strettamente legata all'ereditarietà. Con questo termine si indica la capacità dei metodi delle classi derivate (figlie), di svolgere operazioni diverse a seconda della classe di cui sono istanze.

Spieghiamo meglio con un esempio.

Da una classe Figura_Geometrica è possibile derivare diverse classi figlie, ad esempio Rettangolo, Cerchio, Trapezio.



Le 3 classi figlie ereditano dalla classe madre il metodo (funzione) Calcola_Area, perché vogliamo realizzare un'applicazione che consenta di effettuare tale calcolo per qualsiasi figura geometrica. Poiché il metodo che calcola l'area è comune a tutte le figure geometriche, esso è un metodo della classe madre.

Per ciascuna figura geometrica, però, l'area si calcola in modo diverso, quindi quando si definisce ciascuna delle tre sottoclassi, il codice del metodo Calcola_Area verrà "scritto" dal programmatore in maniera diversa.

Ipotizziamo ora di istanziare 2 oggetti:

- figura1 istanza della classe Rettangolo (figura1 = new Rettangolo)
- figura2 istanza della classe Cerchio (figura2=new Cerchio)

(figura1 e figura 2 è come se fossero due variabili, una di tipo cerchio ed una di tipo rettangolo)

Se operassimo nel'ambito della programmazione procedurale e volessimo calcolare l'area della figura 1 dovremmo scrivere, nel main un codice "simile" a:

```
if (figura1==rettangolo)
        area= calcolaAreaRettangolo(parametri..)
else if (figura1==cerchio)
        area=calcolaAreaCerchio(parametri..)
else
```

```
area=calcolaAreaTrapezio(parametri…)
```

Dove le 3 funzioni di calcolo dell'area sono diverse parti di codice all'interno del programma.

Grazie al polimorfismo, nella programmazione ad oggetti, è possibile semplicemente invocare il metodo calcolaArea sull'oggetto figura1:

```
area= figura1.calcolaArea()
```

l'esecutore "riconoscerà" automaticamente, durante l'esecuzione a runtime che figura1 è un rettangolo, di conseguenza andrà ad eseguire il codice del metodo caclocaArea della classe Rettangolo.

Sembra un vantaggio da poco, ma pensandoci meglio si può osservare che la stessa sequenza di if..else… andrebbe scritta per ciascuna operazione da svolgere su una figura geometrica, ad esempio per calcolare il perimetro:

```
if (figura1==rettangolo)
        perimetro= calcolaPerimetroRettangolo(…parametri)
else if (figura1==cerchio)
        perimetro=calcolaPerimetroCerchio(parametri..)
else
        perimetro=calcolaPerimetroTrapezio(parametri…)
```

Nel main ci sarebbero quindi molti if…else…, necessari per selezionare la funzione corretta a seconda del tipo di oggetto (cerchio, trapezio, rettangolo).

Nel caso in cui si volesse aggiungere al programma la possibilità effettuare calcoli su nuove figure geometriche (ad esempio triangolo, pentagono, esagono, ecc….), oltre a scrivere le funzioni calcolaAreaTriangolo, calcolaPerimetroTriangolo()….nel main sarebbe necessario aggiungere, a tutte le strutture di selezione presenti, il caso

```
if (figura1==triangolo)
```

con grande dispendio di energie da parte del programmatore e con il rischio di dimenticarsi di aggiungere il caso "triangolo" in alcune parti del codice.

Grazie al polimorfismo della OOP, basta creare la nuova classe derivata Triangolo e ridefinire al suo interno i metodi CalcolaArea() e CalcolaPerimetro() con il codice corretto per quella figura geometrica. Infatti nel momento in cui venisse istanziato un oggetto di classe triangolo, il calcolo dell'area e del perimetro si ottiene senza modificare alcuna riga di codice nel main.

```
area= figura1.calcolaArea()
```

funziona qualunque sia la classe di cui è istanza l'oggetto figura1;

Il polimorfismo deriva dalla possibilità, nelle classi derivate, di ridefinire il comportamento dei metodi derivati dalla classe madre.

Classi astratte

Un esempio di **classe astratta** è proprio la classe FiguraGeometrica dell'esempio precedente. **Le classi astratte sono classi che servono solamente per generare classi figlie, ma da cui non si possono istanziare oggetti**. I metodi di queste classi sono solamente delle "signature", come delle funzioni senza body, il cui body verrà scritto nei corrispondenti metodi delle classi figlie. Con riferimento all'esempio precedente, il metodo CalcolaArea della classe Figura_Geometrica non può avere un codice (non può sapere come calcolare l'area di una figura geometrica senza sapere di quale figura si tratta!), è solamente un "segnaposto" per garantire che il body di tale metodo verrà definito, in maniera diversa, in ciascuna sottoclasse corrispondente ad una figura geometrica diversa. Non ha senso quindi istanziare un oggetto della classe FiguraGeometrica, tale classe ha come unico scopo permettere di definire le classi figlie, in cui i metodi sono realmente implementati con del codice.