



Pattern Recognition
and Applications Lab

Le basi della Programmazione Orientata agli Oggetti

Docente: Ambra Demontis

Anno Accademico: 2020 - 2021

Corso di Laurea in Ingegneria Elettrica, Elettronica e Informatica



University of Cagliari,
Italy

Department of Electrical and
Electronic Engineering



Il Paradigma di Programmazione Orientata agli Oggetti

Questo paradigma è ispirato alla realtà.

Problema: vogliamo accendere la televisione.

Soluzione: utilizziamo il telecomando per accendere il televisore.

Diversi problemi “pratici” si risolvono utilizzando oggetti, ognuno con le proprie caratteristiche, e facendoli interagire tra loro.

I linguaggi di programmazione ad oggetti modellano un programma come un insieme di oggetti che interagiscono tra loro.

Oggetto

Entità con degli **attributi** che può, eventualmente, compiere azioni (**metodi**).
Specifica **istanza** di classe e.g., della classe “telecomando auto”.



attributi {
Altezza: 5 cm
Larghezza: 3cm
Spessore: 1 cm

metodi {
-Inserisci antifurto
-Rimuovi antifurto

Classe

Definisce gli attributi e i metodi condivisi da un insieme di oggetti.

Tipo di oggetto: telecomando auto

Attributi:

Altezza

Larghezza

Spessore

Metodi:

Inserisci antifurto

Rimuovi antifurto

Classe vs Oggetto

Diversi oggetti appartengono alla stessa classe se hanno gli stessi attributi e metodi.

Esempio di oggetti che appartengono alla stessa classe:



Appartengono alla stessa classe ma sono oggetti differenti e lo sarebbero anche se fossero all'apparenza identici. Provate ad aprire una macchina con il telecomando di un'altra macchina dello stesso modello...

Le Fasi di Sviluppo di un Programma

- 1. Object-Oriented Analysis:** si ragiona sul problema da risolvere e si *identificano* gli oggetti, le loro proprietà e le interazioni tra gli stessi
- 2. Object-Oriented Design:** si crea un prototipo, cioè si schematizza l'idea derivante dall'analisi al passo precedente
- 3. Object-Oriented Programming:** si implementa il prototipo nel linguaggio di programmazione orientato agli oggetti scelto

Descrivere gli Oggetti: I Diagrammi di Classe

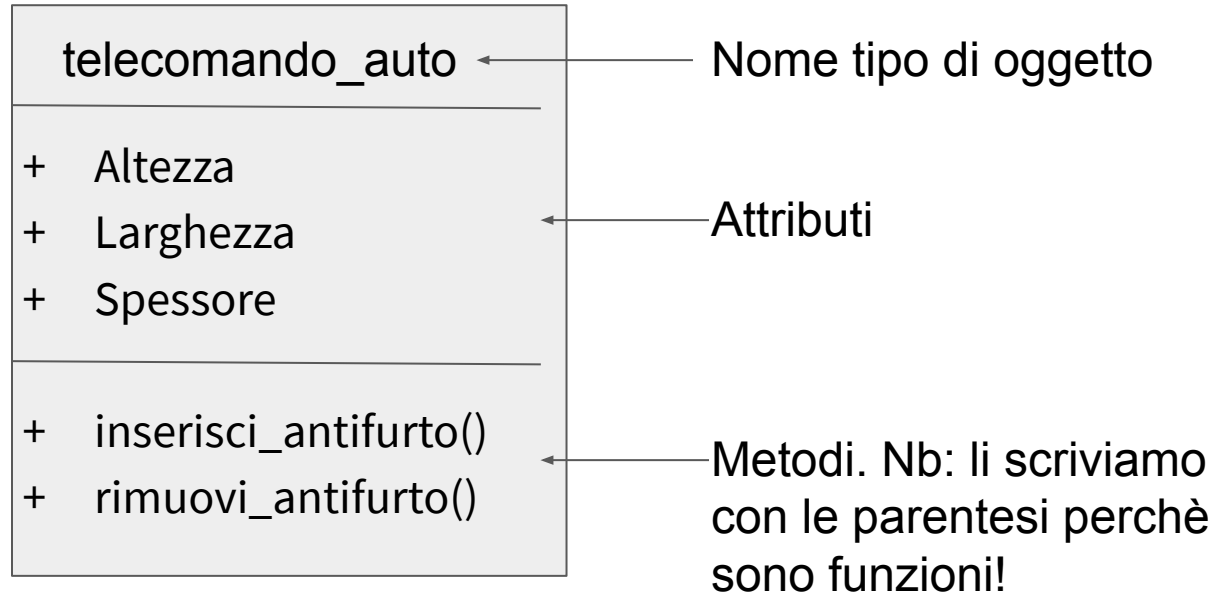
Per schematizzare il prototipo di un programma utilizzeremo i **diagrammi di classe** nel linguaggio “Unified Modeling Language” (UML).

I diagrammi di classe descrivono le classi degli oggetti che compongono il sistema e le relazioni tra essi.

Nb: in queste slide vedremo la notazione mano a mano che ci servirà e verrà poi riepilogata alla fine.

Rappresentare una Classe

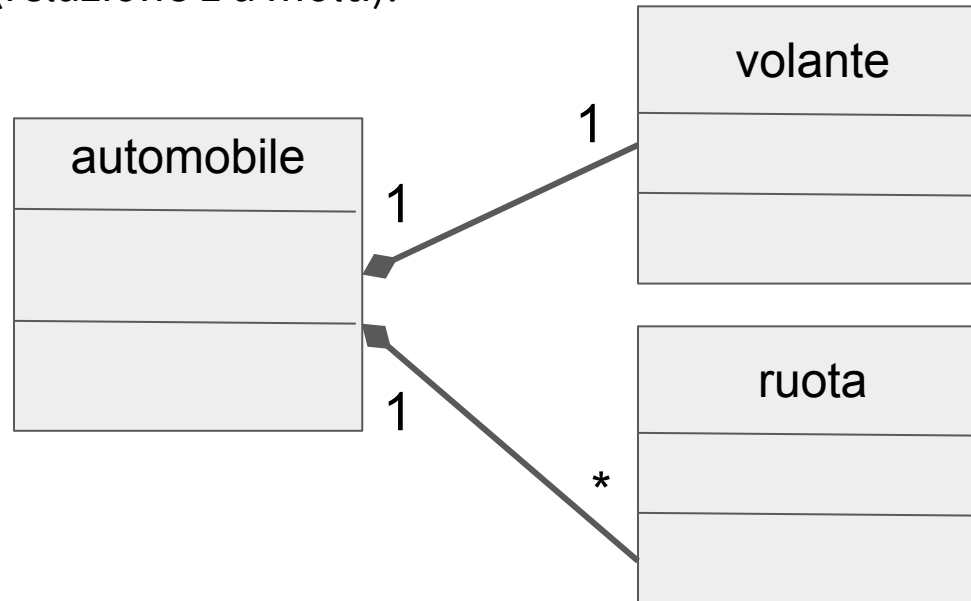
Rappresentiamo la classe *telecomando auto* con un diagramma di classe.



Le Relazioni tra Oggetti: Composizione

Alcuni oggetti possono essere composti da altri oggetti.

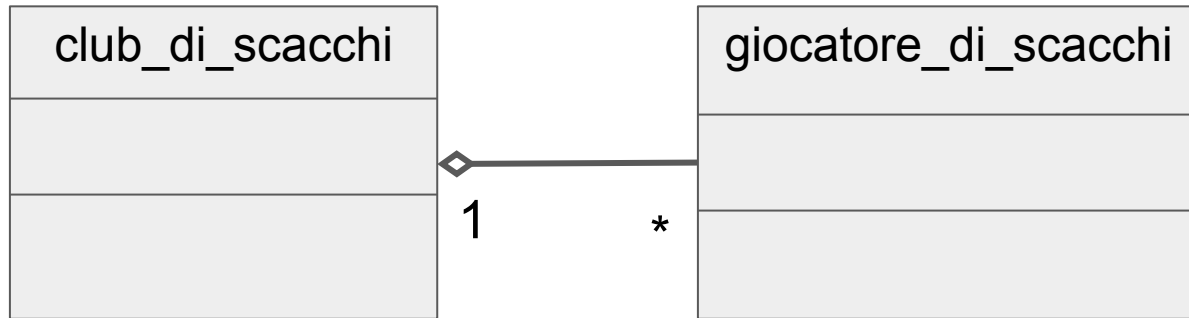
Ad esempio, un'*automobile* è composta da un *volante* (relazione 1 a 1) e da diverse *ruote* (relazione 1 a molti).



Nb: * = molti

Le Relazioni tra Oggetti: Aggregazione

Esistono degli oggetti che aggregano altri oggetti che hanno una vita propria.



Composizione vs Aggregazione

Se distruggo l'oggetto composito (e.g., la *macchina*), distruggo anche gli oggetti che lo compongono (*volante e ruote*).

Si dice che è una relazione “forte”.



Se distruggo l'oggetto che aggrega altri oggetti (e.g., il *club_di_scacchi*) non distruggo gli oggetti che sono aggregati da quell'oggetto (gli oggetti di classe *giocatore_di_scacchi*).

Si dice che è una relazione “debole”.



Le Relazioni tra Oggetti: Ereditarietà

Gli esseri umani ereditano alcune delle loro caratteristiche dai loro antenati.
Anche tra oggetti esiste l'ereditarietà!

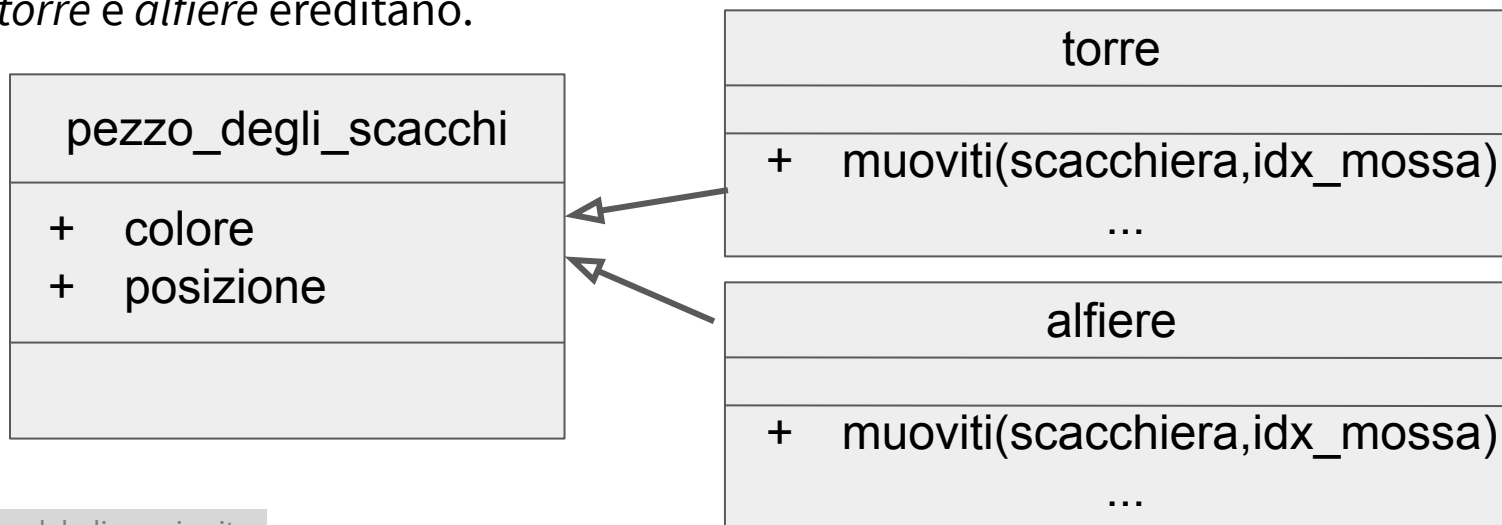
Questo ci permette di evitare la duplicazione di codice nelle classi coinvolte in questa relazione.

Le Relazioni tra Oggetti: Ereditarietà



Un esempio di ereditarietà. I pezzi degli scacchi *torre* e *alfiere* hanno entrambi la caratteristica *colore* (bianco o nero) e entrambi possono effettuare mosse, ma le mosse che possono effettuare sono differenti.

Avremo quindi una classe generica *pezzo_degli_scacchi* dalla quale le classi *torre* e *alfiere* ereditano.

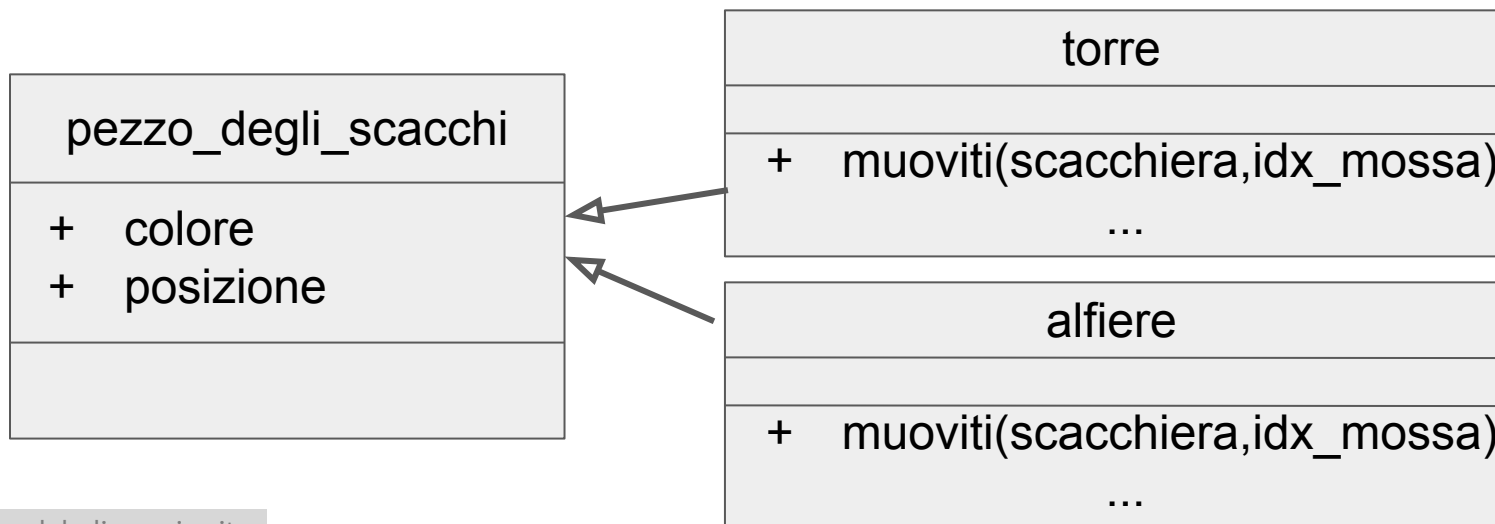


Le Relazioni tra Oggetti: Ereditarietà



La classe pezzo degli scacchi si dice classe **padre** e torre e alfiere sono i **figli**.
I figli ereditano tutto dal padre.

Es: negli oggetti *torre* e *alfiere* non specifichiamo gli attributi *colore* e *posizione* in quanto li ereditano dalla classe padre.

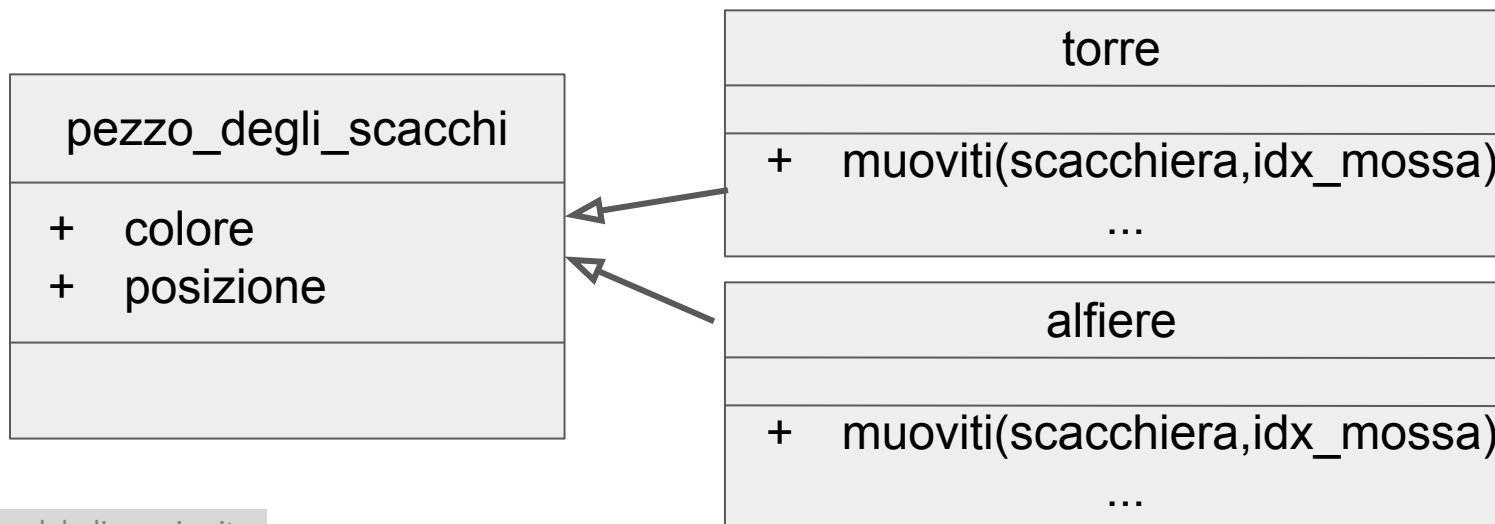


Le Relazioni tra Oggetti: Ereditarietà



I figli ereditano tutto dal padre.

Questo ci permette di evitare di duplicare codice nelle classi coinvolte nell'ereditarietà.

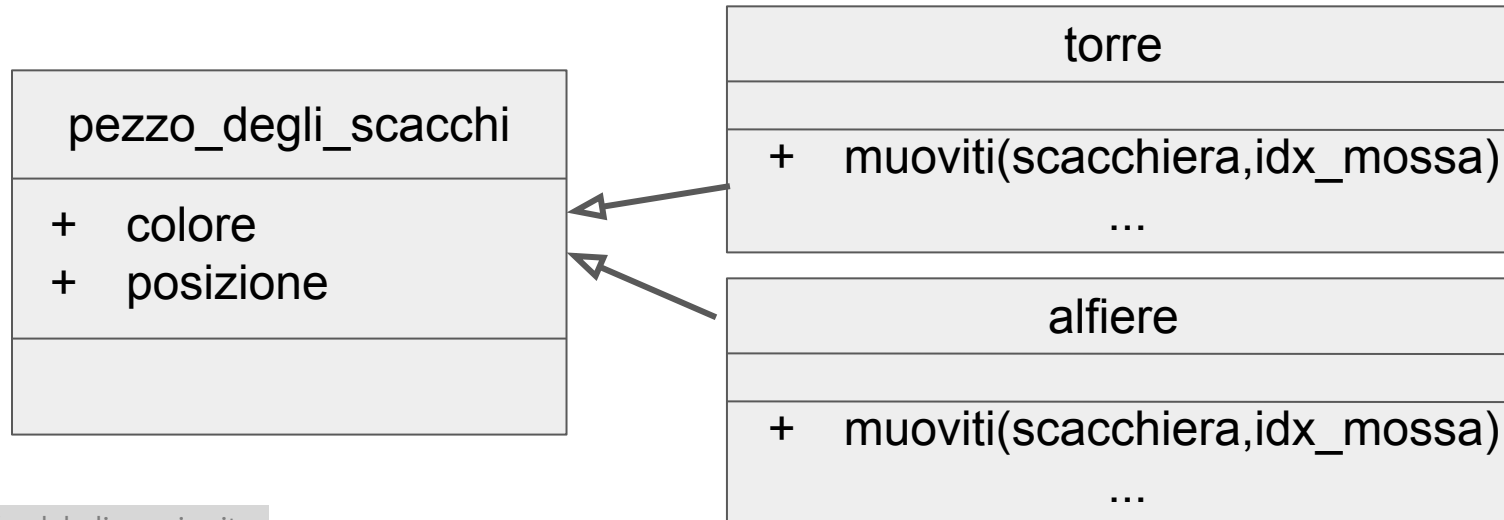


Interfacce Comuni per Oggetti Differenti



La funzione *muoviti* la definiamo nelle classi *torre* e *alfiere* in quanto non tutti i pezzi degli scacchi si muovono nello stesso modo.

Tuttavia per ridurre il codice che le classi utilizzatrici dovranno scrivere, le definiamo con la stessa **interfaccia** (nome di funzione e dei parametri).



Interfacce Comuni per Oggetti Differenti



Vediamo lo pseudocodice (la descrizione a parole del codice) della funzione *muoviti* della classe *torre*:

funzione *muoviti*(scacchiera, idx_mossa):

 se idx_mossa è 1

muoviti_a_destra(scacchiera)

 se idx_mossa è 2

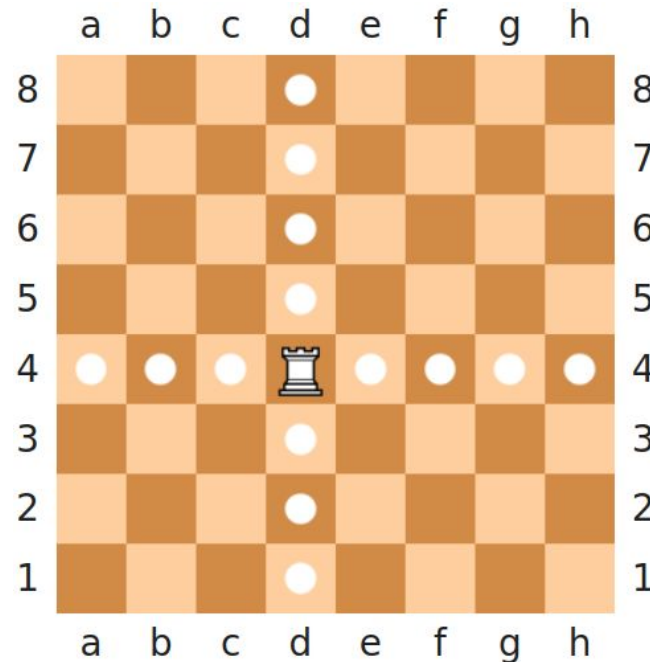
muoviti_a_sinistra(scacchiera)

 se idx_mossa è 3

muoviti_in_alto(scacchiera)

 se idx_mossa è 4

muoviti_in_basso(scacchiera)

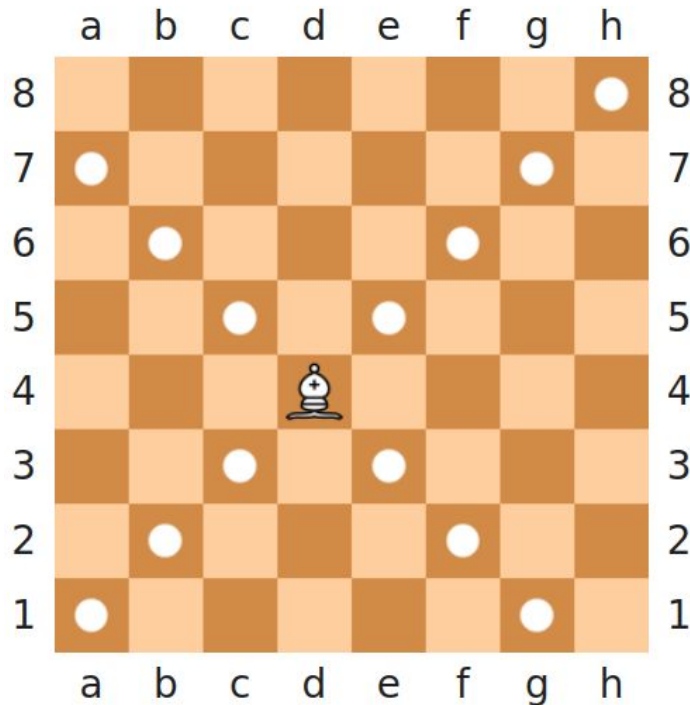


Interfacce Comuni per Oggetti Differenti



Vediamo lo pseudocodice della funzione *muoviti* della classe *alfiere*:

```
funzione muoviti(scacchiera, idx_mossa):  
    se idx_mossa è 1  
        muoviti_a_nord-est(scacchiera)  
    se idx_mossa è 2  
        muoviti_a_nord-ovest(scacchiera)  
    se idx_mossa è 3  
        muoviti_a_sud-est(scacchiera)  
    se idx_mossa è 4  
        muoviti_a_sud-ovest(scacchiera)
```

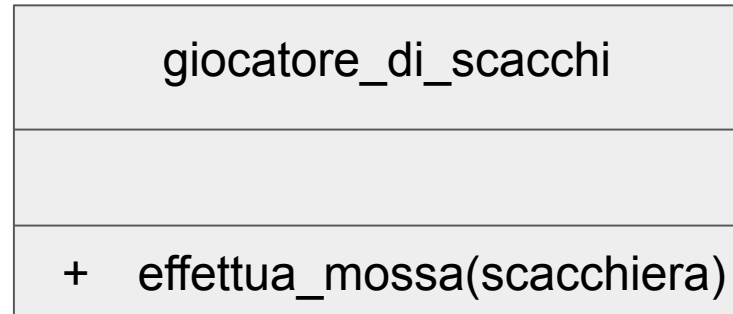


Interfacce Comuni per Oggetti Differenti



Questo ci permette di semplificare il codice nelle classi che utilizzano gli oggetti figli *torre* e *alfiere*.

Proviamo a modellare la classe *giocatore_di_scacchi* che usa i pezzi degli scacchi.



Interfacce Comuni per Oggetti Differenti



funzione `effettua_mossa(scacchiera)`:

`pezzo = scegli_pezzo_da_muovere(scacchiera)`

`idx_mossa = scegli_mossa_da_effettuare(scacchiera)`

`pezzo.muoviti(scacchiera, idx_mossa)`

...

Chiamata la funzione `muoviti` di un oggetto appartenente alla classe `pezzo` degli scacchi.

Il fatto che l'interfaccia della funzione che muove il pezzo sia uguale per tutte le classi figlie di *pezzo_degli_scacchi* semplifica il codice della funzione *effettua_mossa* della classe *giocatore_di_scacchi*! Non deve prevedere diversi casi a seconda del pezzo che si vuole muovere!

Incapsulamento

Vediamo le classi *torre* e *alfiere* con tutte le loro funzioni.

Prima abbiamo evidenziato la funzione *muoviti* che viene utilizzata dagli oggetti di classe *giocatore_di_scacchi* che utilizzano queste classi.

torre
+ muoviti(scacchiera,idx_mossa)
...

alfiere
+ muoviti(scacchiera,idx_mossa)
...

Incapsulamento

Come abbiamo visto dallo pseudocodice però, la classe *torre* e *alfiere* hanno anche delle funzioni che utilizzano per effettuare la funzione *muoviti*. Queste funzioni sono differenti a seconda della classe (torre o alfiere) e non vengono utilizzate dall'oggetto *giocatore_di_scacchi*.

Rappresentiamole nel diagramma delle classi..

Incapsulamento

torre
+ muoviti(scacchiera,idx_mossa) - muoviti_a_destra(scacchiera) - muoviti_a_sinistra(scacchiera) - muoviti_in_alto(scacchiera) - muoviti_in_basso(scacchiera)

alfiere
+ muoviti(scacchiera,idx_mossa) - muoviti_a_nord-est(scacchiera) - muoviti_a_nord-ovest(scacchiera) - muoviti_a_sud-est(scacchiera) - muoviti_a_sud-ovest(scacchiera)

Le funzioni che devono:

- essere utilizzate da altre classi vengono dette **pubbliche** (+)
- essere usate solo dalla classe stessa vengono dette **private** (-)

Il fatto che siano pubbliche o private viene chiamato **visibilità**.

Incapsulamento

Il fatto di “nascondere” attributi e funzioni utili alla classe stessa ma che non sono necessari ad altre classi per utilizzare l’oggetto viene detto **incapsulamento**.

Perchè l’incapsulamento è utile?

Per chi vuole utilizzare la classe è subito chiaro quali funzioni è previsto utilizzi e quindi quali possono essere di suo interesse.

Nei programmi complessi permette di far risparmiare parecchio tempo agli utilizzatori!

Esempi

Il Programma per la Gestione di una Libreria

Supponiamo di essere un programmatore al quale il gestore di una libreria ha chiesto di creare un programma gestionale.

Il programma deve:

- permettere alla libreria di memorizzare i dati di tutti i libri che possiede
- mostrare il prezzo di vendita di un libro considerando che la libreria ha deciso di applicare, a tutti i libri pubblicati da più di 5 anni uno sconto del 5%.

Object-Oriented Analysis

Abbiamo un problema da risolvere con tecniche di OOP.

Come abbiamo detto precedentemente il primo passo è capire quali classi ci servono e quali devono essere le loro proprietà e i loro metodi.

Quali Classi ci servono?

Quali Classi ci servono?

- 1) dobbiamo memorizzare i dati di ogni libro, memorizzare il prezzo iniziale e calcolare il prezzo scontato, quindi ci servirà una classe *libro*
- 2) Dobbiamo memorizzare e cercare i dati di più libri quindi ci servirà una classe che raccolga gli oggetti di classe libro es. la classe *lista_libri*

La Classe Libro

Deve permetterci di memorizzare i dati del libro...

libro
+ titolo
+ autore
+ Editore
+ prezzo
+ anno di pubblicazione

La Classe Libro

Deve anche permetterci di calcolare il prezzo scontato del libro.

libro
+ titolo
+ autore
+ editore
+ prezzo
+ anno di pubblicazione
+ calcola_prezzo_scontato()

Per poter calcolare il prezzo scontato, dobbiamo calcolare l'età del libro e calcolare lo sconto. E' quindi comodo creare due funzioni private...

La Classe Libro

Deve anche permetterci di calcolare il prezzo scontato del libro.

libro
+ titolo + autore + editore + prezzo + anno di pubblicazione
+ calcola_prezzo_scontato() - calcola_eta_libro() - calcola_sconto()

Lista Libri

Deve permetterci di memorizzare e cercare i libri.

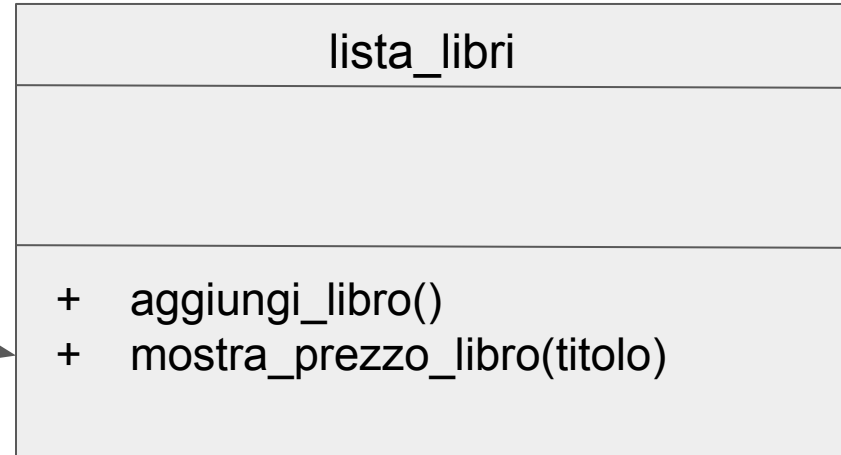
Crea un oggetto di tipo libro
E lo memorizza



Lista Libri

Deve permetterci di memorizzare e cercare i libri.

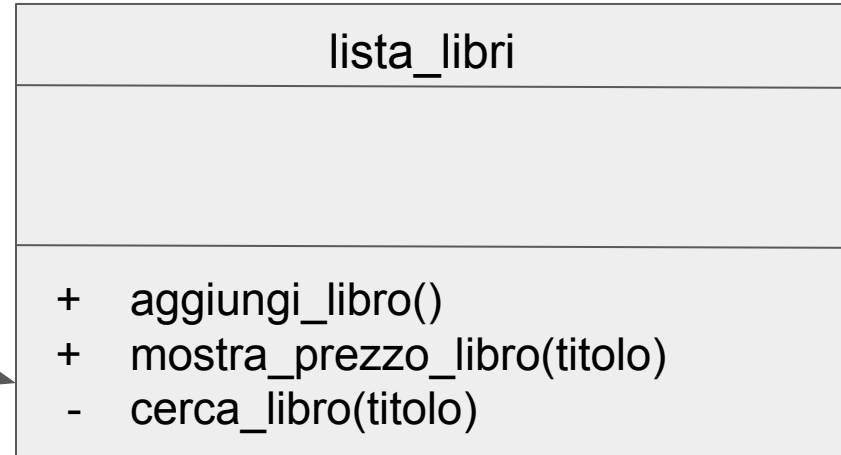
Funzione pubblica che cerca il libro, calcola il prezzo scontato (richiamando l'apposita funzione della classe libro) e mostra a schermo il prezzo del libro



Lista Libri

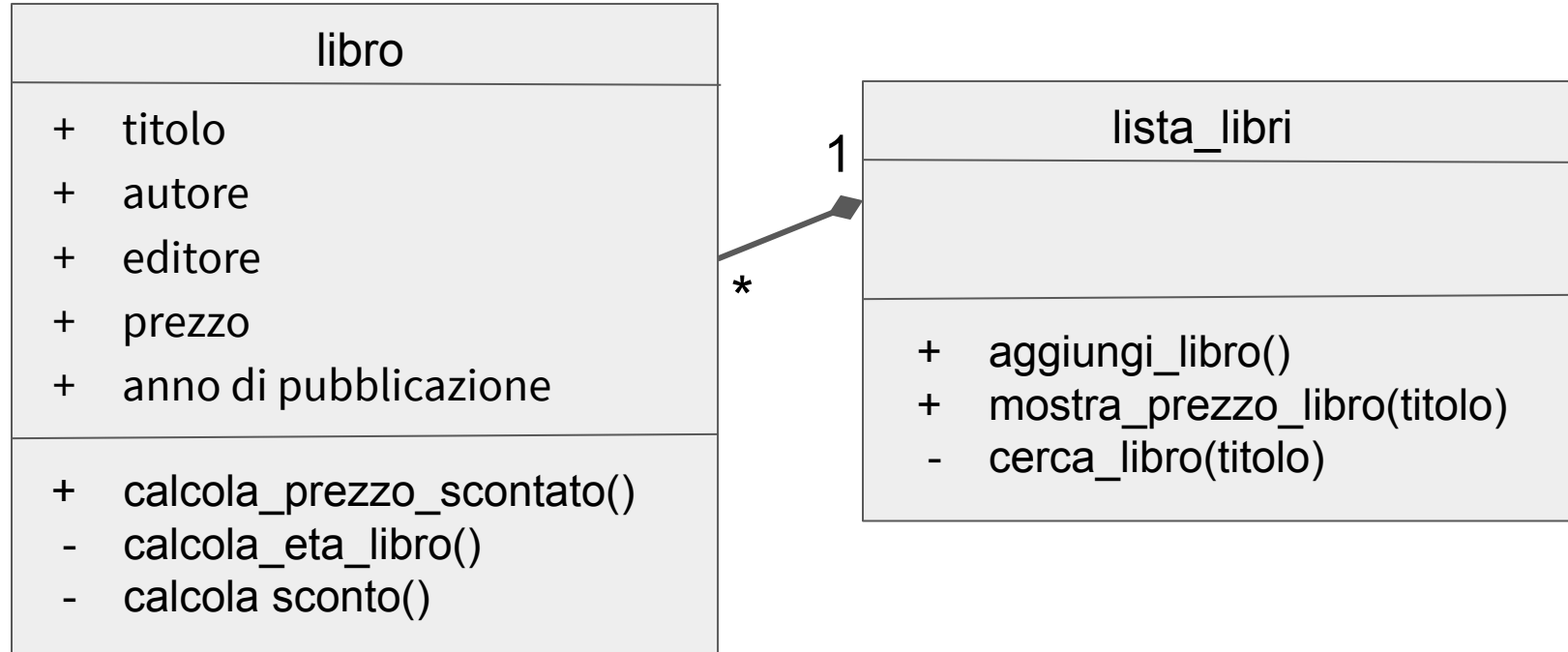
Deve permetterci di memorizzare e cercare i libri.

Funzione privata che dato un titolo
cerca un libro e lo restituisce (utilizzata
dalla funzione pubblica
mostra_prezzo_libro)






Lista Libri

La lista libri aggrega oggetti di classe libro.



Riepilogo Notazione

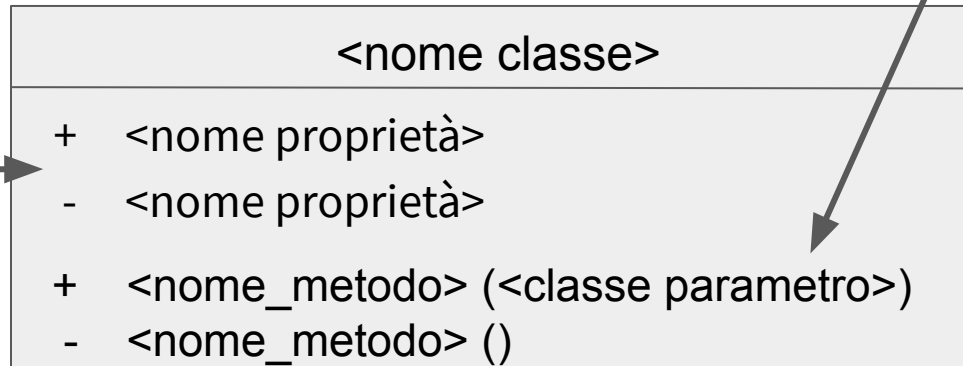
Relazioni:

- Composizione 
- Aggregazione 
- Ereditarietà 

Rappresentazione oggetti:

Visibilità

- + pubblico
- privato



Metodo che riceve
un parametro

Riepilogo Notazione

Volendo è possibile specificare anche i tipi di dato.

Tipo di dato della
proprietà es:
+ nome: stringa

<nome classe>	
+ <nome proprietà>: tipo_dato	
- <nome proprietà>: tipo_dato	
+ <nome_metodo> ():tipo_dato	
- <nome_metodo> (<parametro>: tipo_parametro): tipo_dato	

Tipo di dato restituito
dalla funzione
void se non
restituisce nulla

Esercizi

Voto Finale Esami

Progettiamo un programma che permetta ad uno studente di memorizzare, per ogni esame, i voti presi in due esami parziali e che gli permetta di stampare il voto finale conseguito in tutti gli esami.

Di quali Oggetti Abbiamo Bisogno?

Di quali Oggetti Abbiamo Bisogno?

- 1) Un oggetto che contenga i voti di un esame
- 2) Un oggetto che ci permetta di memorizzare i voti di tanti esami

La classe `Voti_esame`

<code>voti_esame</code>
<ul style="list-style-type: none">+ <code>nome_esame</code>+ <code>voto_primo_parziale</code>+ <code>voto_secondo_parziale</code>
<ul style="list-style-type: none">+ <code>calcola_media()</code>+ <code>stampa_voto()</code>

La classe Lista_voti

lista_voti
+ stampa_voti_esami()

La Relazione tra le classi

