UML e progettazione OOP

Emanuele Ing. Benatti

Unified Modeling Language .

- un linguaggio (e notazione) universale, per rappresentare qualunque tipo di sistema software
- uno standard OMG (Object Management Group), dal nov.1997
- gli autori:
- Grady Booch
- Ivar Jacobson
- Jim Rumbaugh
- i co-proponenti: Microsoft, IBM, Oracle, HP, Platinum, Sterling, Unysis (e tanti altri)

Unified Modeling Language .

- Approccio Object-Oriented che consente di rappresentare:
  - ✔ Struttura statica
  - ✔ Comportamento
  - ✔ Interazioni tra diverse componenti del sistema
- Linguaggio semi-formale
- Notazione grafica
- Strumenti a supporto della progettazione

Unified Modeling Language - Cos'è

- è un linguaggio di progettazione, non un linguaggio di programmazione (come Java, Python, C++, …)
- quindi serve a progettare un nuovo sistema, o a apportare modifiche alla progettazione di un sistema esistente, senza perdersi nei dettagli dei linguaggi di programmazione
- è universale, nel senso che può rappresentare sistemi molto diversi senza differenze legate alla tecnologia: dai sistemi web a quelli più tradizionali, dalle vecchie applicazioni Cobol a quelle object oriented e a componenti

Unified Modeling Language - Cosa non è

- è un linguaggio, non un metodo
- definisce una notazione standard, basata su un metamodello integrato degli “oggetti” che compongono un sistema software
- ma non prescrive una sequenza di processo, cioè non dice “prima bisogna fare questa attività, poi quest’altra”
- quindi può essere (ed è) utilizzato da persone e gruppi che seguono metodi diversi (è “indipendente dai metodi”)

Unified Modeling Language - Tipi di diagrammi

- Casi d’uso (use case),
- Classi (class diagram), identifica le relazioni tra le classi
- Sequenze (sequence diagram), identifica le sequenze di attività
- Attività (activity diagram), tutte le attività che vengono svolte

La progettazione UML ha fasi simili alla progettazione del sw a cascata:
1 – Analisi.

2 – Progettazione di sistema.

3 – Progettazione degli oggetti.

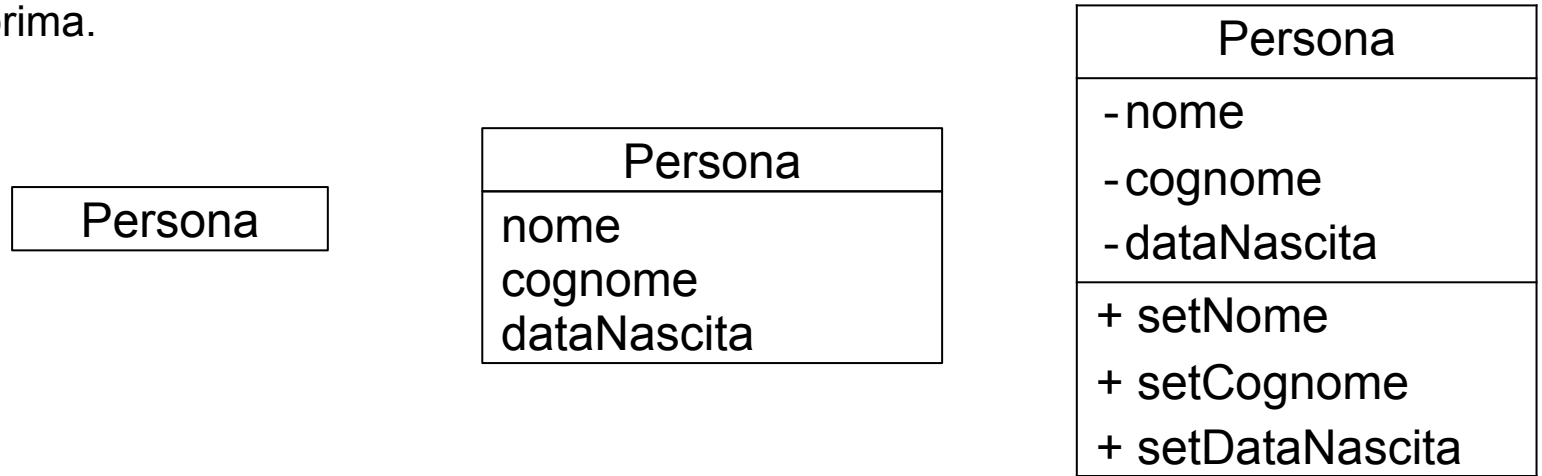
4 – Implementazione.

Fasi progettazione UML (Esempio libreria)

ANALISI: Si costruisce un modello della realtà (es. una libreria)
Viene determinato COSA il sistema deve fare e non COME deve essere fatto (gestione libreria, prestiti, acquisti, giacenze).
Si individuano quali sono:
Gli oggetti su cui lavora il software (libri, ordini, prestiti), gli utenti (bibliotecari, lettori).
PROGETTAZIONE DI SISTEMA: Si definisce in modo progettuale, non quello che il sistema software deve fare.
Si divide il sistema software in sottosistemi che poi saranno realizzati separatamente in modo da dar vita ad un lavoro modulare.
PROGETTAZIONE SW: Si progettano come saranno gli oggetti del software (libri, ordini, prestiti….) cioè le strutture dati (classi, funzioni ecc.)
IMPLEMENTAZIONE: Codifica dell'UML.

Diagramma delle classi

- E' il più comune tra i diagrammi UML
- Descrive le tipologie di oggetti che compaiono in una particolare realtà (o dominio) di interesse (la realtà che si vuole analizzare)
- In altre parole, descrive le classi di oggetti che fanno parte di un sistema e le relazioni tra queste classi.
- Ogni classe è rappresentata da un box suddiviso in tre parti: nome, attributi (o proprietà), metodi (o operazioni). Di queste tre parti, è obbligatorio specificare solo la prima.



OOAD (Object Oriented Analysis and Design)

I diagrammi delle classi possono essere utilizzati per rappresentare il risultato dell'**analisi a oggetti** di un determinato insieme di requisiti.

L'analisi a oggetti ha lo scopo di individuare, ad un livello molto alto di astrazione, quali sono gli oggetti principali della realtà che si sta analizzando (questi non rispecchiano necessariamente le classi della futura applicazione)

Ragionare "a oggetti" significa pensare la realtà in termini di oggetti. Ogni oggetto è dotato di uno stato interno e di un insieme di comportamenti/azioni. Gli oggetti interagiscono tra loro scambiandosi messaggi.

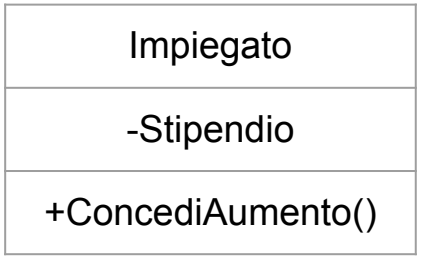
Effettuare la progettazione a oggetti significa individuare le classi principali che formeranno l'applicazione, i loro attributi e metodi principali, le loro relazioni con le altre classi.

Completezza degli schemi

In generale, in tutti i diagrammi UML è sempre consentito **omettere** alcune informazioni allo scopo di concentrarsi solo sulle informazioni più importanti da trasmettere, tralasciando i dettagli

In generale, l'assenza di un'informazione in un diagramma UML non dimostra che quella caratteristica non debba essere presente!

Esempio:



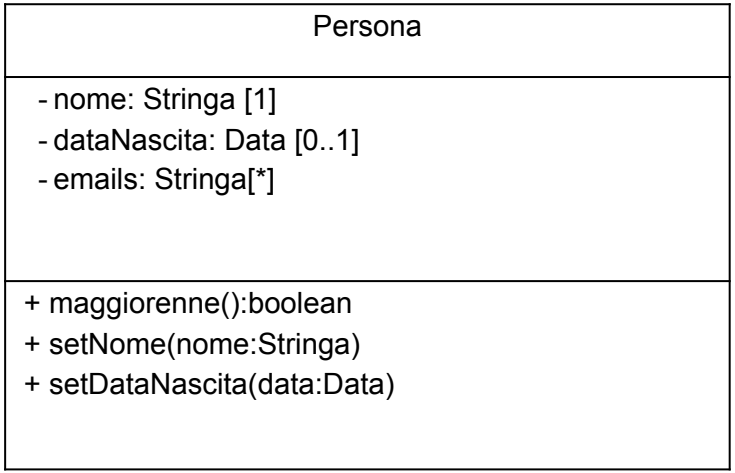
In questo esempio non è detto che Impiegato non abbia altri metodi, ma certamente avrà un metodo ConcediAumento()

Formalismo base

E' possibile specificare tipi di dato e molteplicità per attributi e metodi usando le seguenti notazioni:

visibilità nomeAttributo : tipo molteplicità

visibilità nomeMetodo(listaParametri) : tipoRitorno



L'indicatore di visibilità può essere:

\+ PUBBLICO - PRIVATO # PROTECTED ~PACKAGE

Molteplicità

La molteplicità di un attributo indica quanti valori sono associati all'attributo.

La molteplicità si esprime con due numeri positivi riportati tra parentesi quadre, che indicano rispettivamente la molteplicità **minima** (limite inferiore) e quella **massima** (limite superiore)

Esempio: - telefono: String[1..3]indica che per ogni persona è possibile memorizzare da 1 a 3 numeri di telefono.

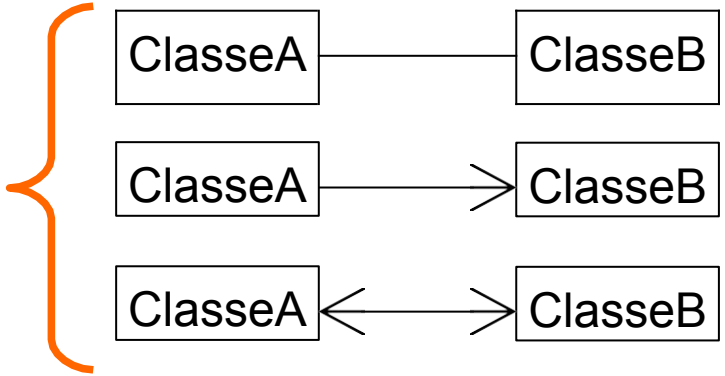
Se il minimo e il massimo coincidono, è possibile scrivere un solo numero.
Quindi [1] è sinonimo di [1..1].
[*] indica [0..N] , con N imprecisato.

Se il limite inferiore è zero si dice che l'attributo è opzionale, altrimenti è obbligatorio.

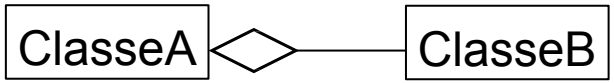
Se il limite superiore è 1 si dice che l'attributo è **a un solo valore**.

Relazioni tra classi

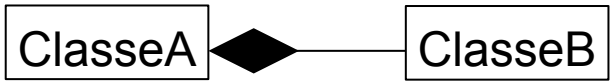
- **Associazioni**
  - Associazione generica



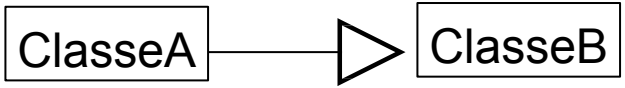
  - Aggregazione



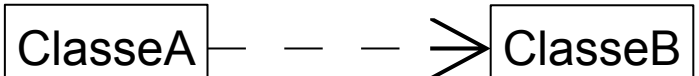
  - Composizione



- **Generalizzazione**



- Dipendenza



Associazioni

Un'associazione indica che esiste un qualche legame logico generico tra due classi (esempio: una persona **possiede** una o più automobili; uno studente **frequenta** un corso, un giocatore **partecipa** a una partita, ecc..)

L'associazione di indica con una **linea continua**

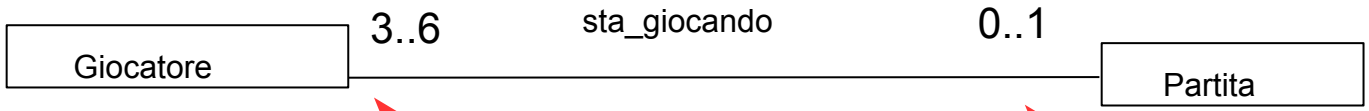
Sulla linea è possibile indicare il nome dell'associazione (in genere un verbo) ed eventualmente un verso di lettura (se non è già chiaro dal contesto)

Molteplicità delle associazioni

è possibile specificare la molteplicità (minima e massima) per ciascun lato di un'associazione

Le molteplicità pongono dei **vincoli** sul numero di oggetti dello stesso tipo che possono partecipare a un'associazione

Esempio: *in un'ipotetica applicazione di gioco online, per avviare una partita occorrono almeno 3 giocatori collegati, e ad una certa partita non possono giocare più di 6 giocatori. Ogni giocatore non può partecipare a più partite contemporaneamente.*



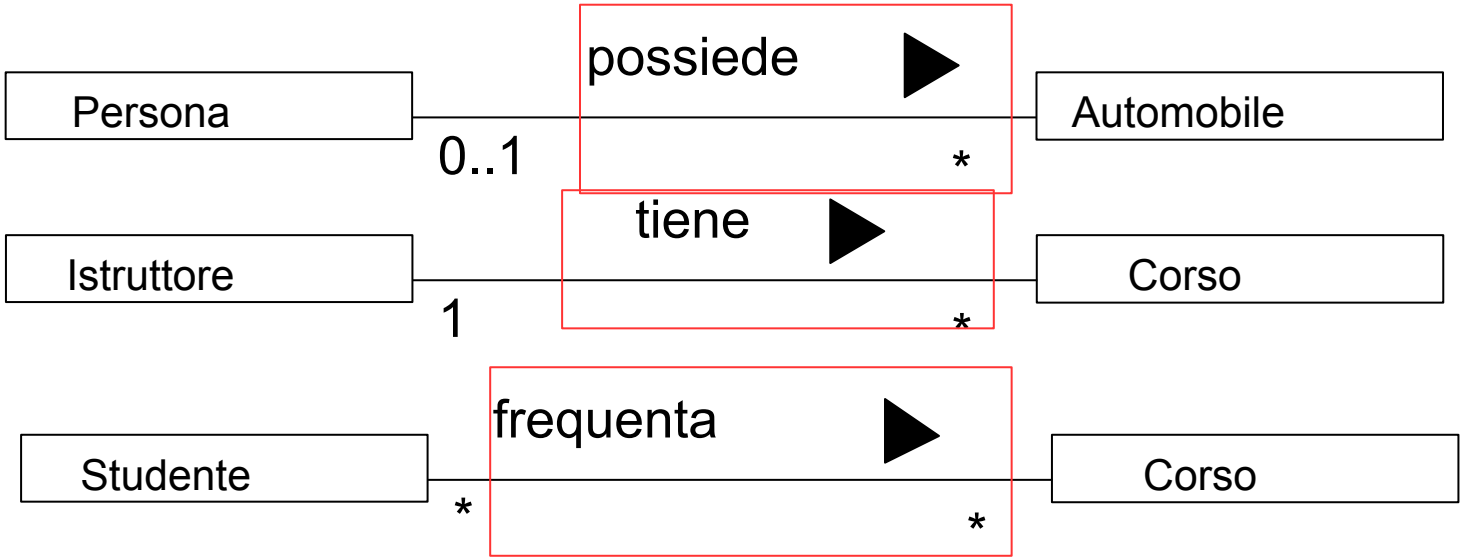
Quante istanze di Giocatore sono collegate ad una Partita? Da 3 a 6

Quante istanze di Partita sono collegate ad un Giocatore? Da 0 a 1

Molteplicità delle associazioni

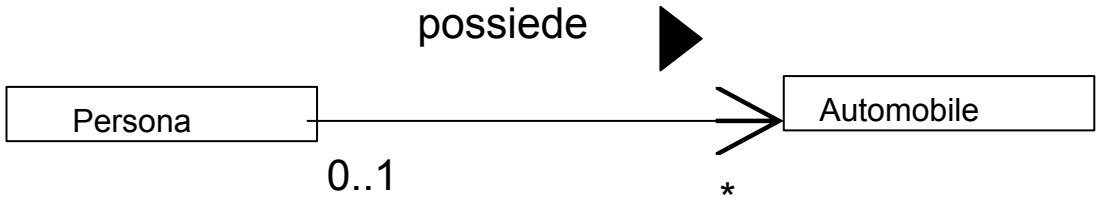
Valgono le abbreviazioni già viste:

- 1 equivale a 1..1
- * equivale a un numero qualunque compreso tra 0 e infinito

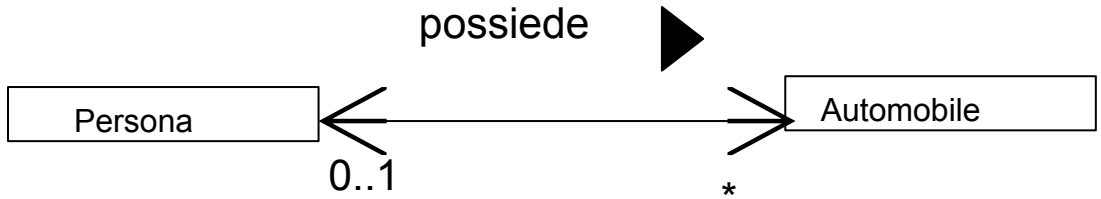


Navigabilità

Ogni persona possiede un numero qualunque di auto ma un auto può avere un solo proprietario



Associazione monodirezionale

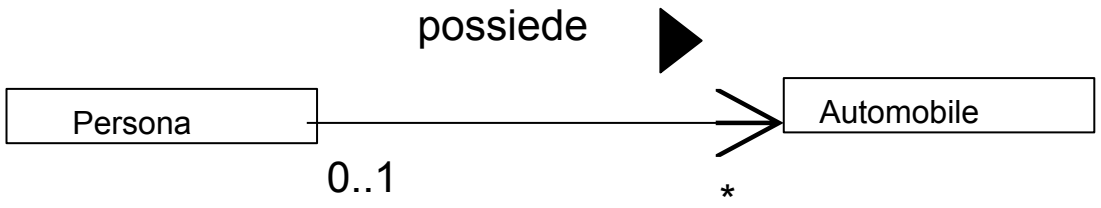


Associazione bidirezionale

In corrispondenza di uno solo o di entrambi gli estremi di un'associazione è possibile disegnare una freccia come quelle mostrate in figura, che indica in quali versi è possibile "muoversi" (navigare) tra i vari oggetti

Una freccia di navigabilità rivolta dalla classe A verso la classe B indica che a partire da una certa istanza di A si può risalire alle istanze collegate della classe B.

Implementazione

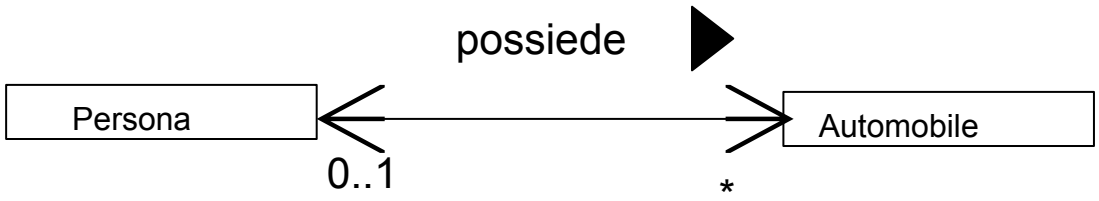


Associazione monodirezionale

```
class Persona {

    private String nome;
    private String codiceFisc;
    private Automobile[] auto;

}

class Automobile {
   private String targa;
   private String modello;

}
```

E' comunque consigliato NON mettere frecce se non si è certi della direzionalità o la direzionalità è bidirezionale.
E' consigliabile tenere il concetto di freccia quasi esclusivamente per il concetto di gerarchia!

Implementazione

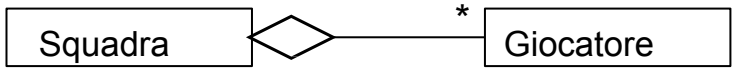


Associazione bidirezionale

```
class Persona {
    private String nome;
    private String codiceFisc;
    private Automobile[] auto;
}
class Automobile {
    private String targa;
    private String modello;
    private Persona proprietario;

}
```

Aggregazione

- L'associazione esprime un legame logico "generico"
- quando l'associazione assume il significato di **parte di** è possibile usare il simbolo di **aggregazione** (rombo vuoto)
  Esempio

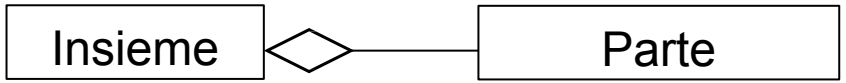


- Un giocatore è un componente (una parte) della squadra. La squadra è un insieme di giocatori (è composta da giocatori in questo caso uno o più giocatori).

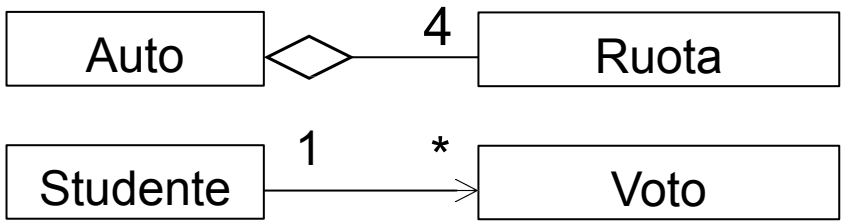
  Un oggetto A è parte dell'oggetto B;

Aggregazione

L'aggregazione quindi esprime il concetto di "Insieme" o "Aggregato" e "Parti" o "Componenti"



Non tutte le associazioni esprimono questo concetto. Esempi:

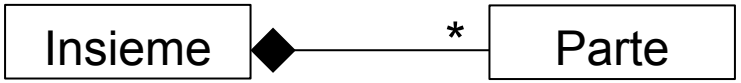


Uno studente non è "composto da" voti

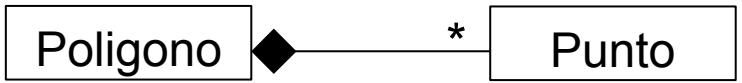
Tuttavia, la classe Studente è composta da un insieme di voti. In definitiva, UML non definisce chiaramente la semantica dell'aggregazione, per cui è possibile usarla o meno il rombo vuoto in modo piuttosto "libero". Se presente, serve a rafforzare il significato di "A è composto da B" o equivalentemente "B è un componente di A".

Composizione

La composizione indica un legame tutto-parte (o insieme-componenti). La composizione però indica un **possesso più forte.**

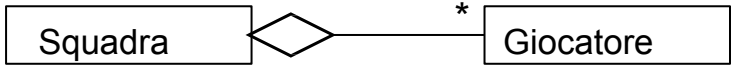


Una relazione di composizione tra la classe A (il "tutto") e la classe B (la "parte") indica che le **istanze** di B **non hanno senso di esistere al di fuori dell'istanza di A che le contiene**. Quindi, cancellando l'istanza di A è corretto perdere anche gli oggetti della classe B in essa contenuti.

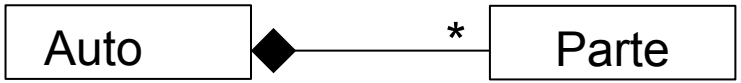


Aggregazione vs. Composizione

La composizione indica un legame tutto-parte (o insieme-componenti). La composizione però indica un **possesso più forte.**



Aggregazione (legame debole)
Un giocatore esiste anche se non è legato ad una specifica squadra.

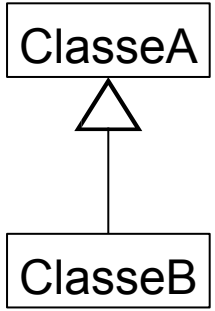


Una parte se non esiste l'auto non ha senso di esistere in quanto non è dotata di autonomia.

Nelle aggregazioni/composizioni, salvo diversa specifica è sempre obbligatorio mettere la molteplicità del lato sinistro (squadra e auto in questo esempio)

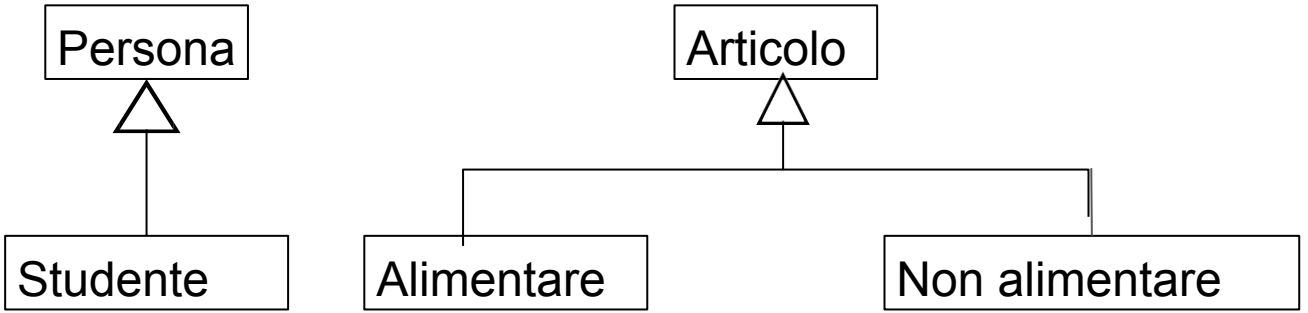
Generalizzazione

La generalizzazione prevede che

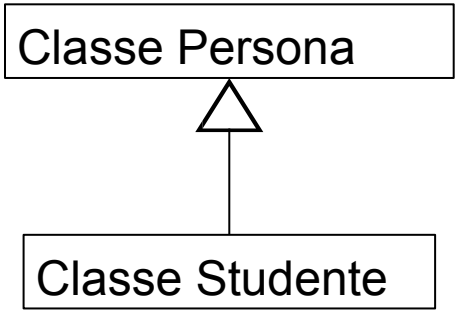


La classe B specializzi la classe A, quindi di fatto tutti gli oggetti di B sono anche oggetti di A.

Esempio



Generalizzazione



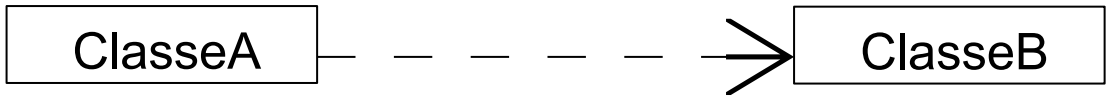
Implementazione

```
class Persona{
     private string nome;
     private string cognome;
     private string ComuneResidenza;
};

class Studente extends Persona{
     private int matricola;
     private int corsoIscrizione;
};
```

Oppure si poteva usare implements, cioè estendere un'interfaccia

Dipendenze

Una freccia tratteggiata dalla classe A alla classe B indica che **la classe A dipende dalla classe B**



Ciò significa che eventuali modifiche alla classe B possono generare la necessità di modifiche anche nella classe A

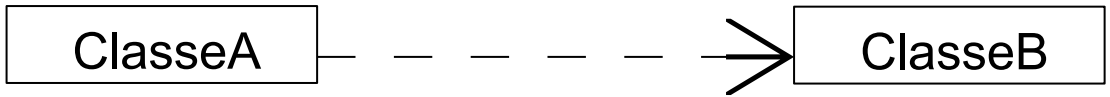
La classe A è detta classe cliente (**client**) e la classe B è detta classe fornitore (**supplier**)

Ogni volta che una classe utilizza un metodo di un'altra classe, la prima classe dipende dalla seconda...

Non ha alcun senso mostrare tutte le possibili dipendenze in un diagramma delle classi (sono troppe!). Si mostrano solo le dipendenze "significative"

OOP e dipendenze

Una freccia tratteggiata dalla classe A alla classe B indica che **la classe A dipende dalla classe B**



Nel paradigma di programmazione OOP, per definizione ogni oggetto, tramite l'interfaccia pubblica interagisce, tramite l'invocazione di metodi con altri oggetti (anche di altre classi).

Di fatto, un programma ben scritto in questo paradigma **DEVE** prevedere chiare interfacce di interoperabilità tra oggetti. Cioè deve essere ben documentato, per ogni metodo cosa esattamente andrà a fare e quali sono i valori restituiti dalla funzione. **Scrivere della buona documentazione diventa quindi fondamentale.**

OOP e dipendenze: esempio

Una freccia tratteggiata dalla classe A alla classe B indica che **la classe A dipende dalla classe B**

Esempio significativo: abbiamo una classe che implementa la logica (Model) di un oggetto all'interno di un magazzino, e abbiamo una classe (view) che implementa l'interfaccia grafica (GUI - Graphical User Interface) che viene implementata per realizzare ciò deve essere visualizzato di quell'oggetto.
Questa dipendenza è importante perchè modificato il Model (oggetto) dobbiamo modificare certamente anche la view (FinestraOggetto)

Il paradigma di questo esempio viene chiamato MVC (Model View Controller)

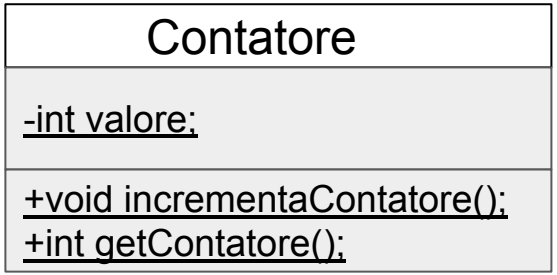
Metodi statici

Si dice statico un attributo o un metodo che appartiene alla classe anziché ad una sua istanza.

In UML i metodi e gli attributi statici vanno sottolineati nel diagramma

Es.

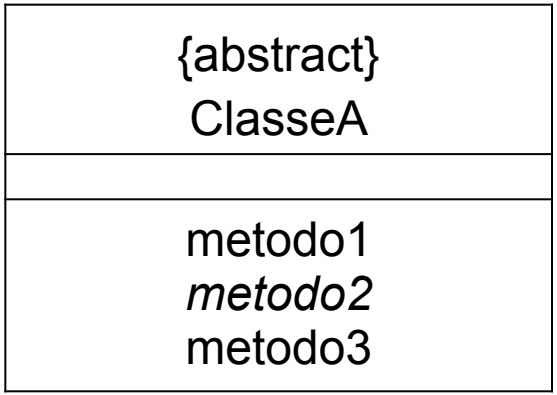
```
class Contatore
{
        private static int valore;
        public static getContatore();
        public static int getContatore();
}
```



Classi e metodi astratti

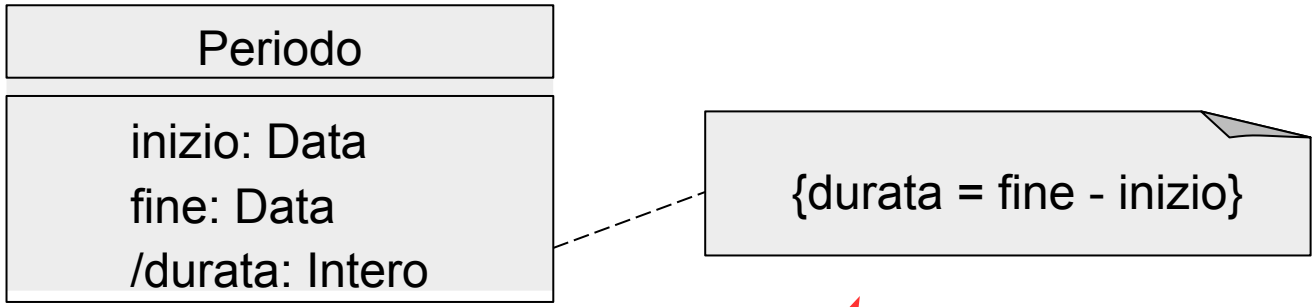
Le interfacce si indicano come le classi (box in cui non vi saranno attribuiti) con l'aggiunta della parola chiave "interface" sopra il nome.

In modo analogo, per le classi astratte si aggiunge la parola chiave "abstract" sopra il nome della classe. Inoltre il nome della classe e il nome dei metodi astratti è scritto in corsivo.



Proprietà note

E' possibile specificare proprietà calcolate in base ad altre.
Le proprietà derivate vanno fatte precedere da / Esempio



è possibile aggiungere delle note collegate alle classe da una linea tratteggiata