Basi di Dati 2

Marco Casu

Contents

1 Introduzione

Questo corso non è ristretto esclusivamente alla progettazione di basi di dati, bensì fornisce cenni sulla progettazione di software di grandi dimensioni, supportati da basi di dati reali.

Un cliente (committente) fornisce delle specifiche riguardo un progetto che bisogna sviluppare, esso stesso non sa come verrà implementato o quali sono nello specifico tutte le funzionalità, un insieme di ingegneri del software, progettisti, e programmatori si occuperanno di "tirare su" il lavoro completo nel tempo, e varie figure professionali verranno necessariamente coinvolte.

Tempi per un progetto software complesso :

- Capire il problema e cosa vuole realmente il cliente : 33% del tempo totale.
- Progettazione, capire come implementare le richieste del cliente : 50% del tempo totale.
- Effettiva realizzazione (sviluppo del codice) : 17% del tempo totale.
- Del tempo extra per i test di verifica e la manutenzione.

1.1 Contesto Organizzativo

Le figure professionali *chiave* coinvolte nel progetto sono dette **attori**, generalmente sono :

- Committente ed Esperti del dominio
- Analisti e Progettisti
- Programmatori
- Utenti finali e Manutentori

Qual'è la differenza tra analisti e progettisti? E di cosa si occupano gli esperti del dominio?

Il **dominio** dell'applicazione è l'insieme di informazioni necessarie da conoscere per poter lavorare ad un progetto che fa riferimento ad uno specifico ambito, ad *esempio*, un applicazione che si occupa di registrare e gestire le contravvenzioni stradali, vedrà sicuramente nel suo dominio il codice stradale e le informazioni legislative.

L'esperto del dominio è una figura, appunta esperta, del dominio inerente al progetto in questione, viene pagata dal committente e funge da consulente durante lo sviluppo.

1.2 Ciclo di Vita del Software

È possibile suddividere lo sviluppo di un software in macro-fasi principali.

1. **Studio di fattibilità** - Ci si approccia al progetto valutando i costi per realizzarlo ed i benefici, si pianificano le attività e le risorse del progetto, umane ed economiche, e si individua l'ambiente di programmazione hardware e software.
2. **Raccolta dei requisiti** - Bisogna capire *cosa il sistema deve fare*, scrivere in prosa una documentazione che descriva precisamente le usabilità del progetto, sintetizzando i requisiti, che spesso sono contraddittori, trovando i giusti compromessi.

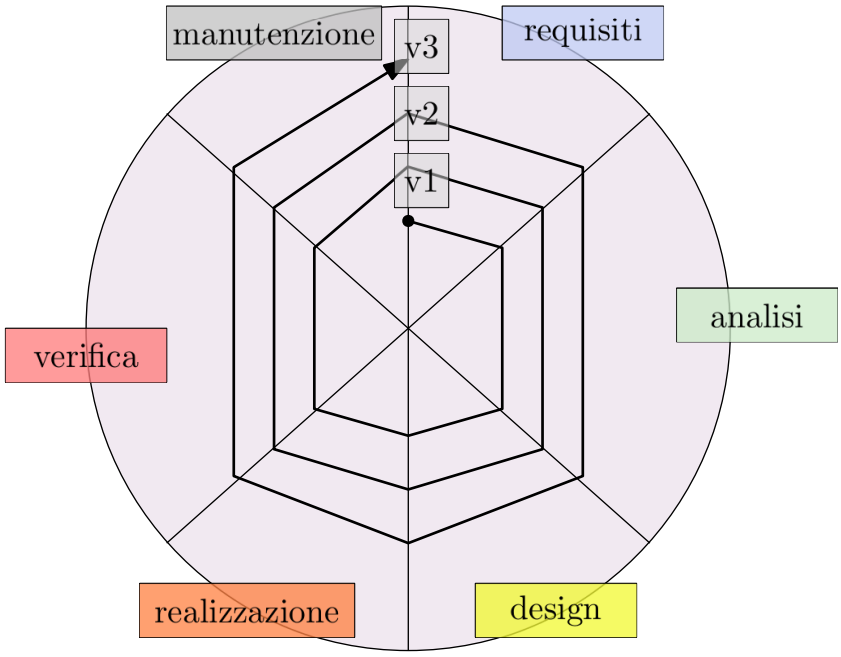
3. **Analisi concettuale dei requisiti** - Sono coinvolti gli analisti, che produrranno uno schema matematico del progetto, dettagliato per filo e per segno, che definirò cosa l'applicazione deve fare indipendentemente dal come. Lo schema prima citato è detto *schema concettuale*, e sarà la base da cui partire per la progettazione.

4. **Progettazione (design) dell'applicazione** - Bisogna capire *come* il sistema realizzerà le sue funzioni, entra in gioco il progettista, che definirà l'architettura volta ad ospitare il software e l'insieme delle tecnologie necessarie.

5. **Realizzazione** - Una volta che si hanno le linee guida per la realizzazione, composte nelle fasi precedenti, si delega la scrittura del codice ai programmatori, che non sono coinvolti nel resto e non devono necessariamente essere a conoscenza di cosa stanno facendo, ma esclusivamente produrre le funzioni richieste.

6. **Verifica, esercizio e manutenzione** - Le diverse componenti dell'applicazione vengono integrate. Una volta che il progetto è realizzato e pronto alla messa in esercizio, si passa da una fase di testing ad una fase di utilizzo effettivo, l'applicazione verrà monitorata durante l'esercizio ed eventuali correzzione verranno prodotte.

Si osservi il seguente diagramma rappresentante il **modello a spirale** di realizzazione :



Tutto il progetto viene costruito in maniera "iterativa", si dice che lo sviluppo del software sia *agile*, si comincia raccogliendo i requisiti strettamente necessari, per poi procedere all'analisi considerando tali requisiti, con l'andare avanti delle fasi portando alla realizzazione di una prima versione del software, pronta ad essere messa in esercizio, implementante esclusivamente le funzionalità di base, tale versione renderà chiare le idee al committente che potrà fornire nuovi requisiti, in modo tale da ricominciare il ciclo.

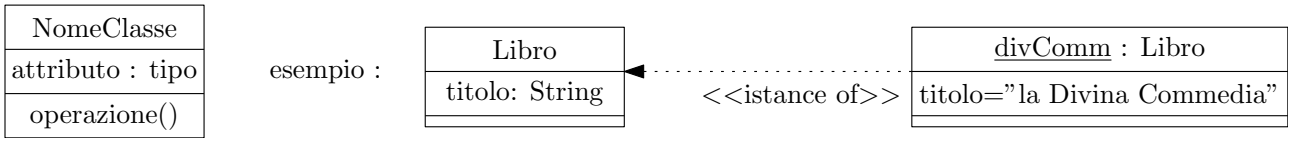
Nulla vieta alle varie fasi di essere eseguite in parallelo, ad esempio, nel tempo t_0 vengono stilati i requisiti per la prima versione del software, nel tempo t_1 gli analisti iniziano a produrre il modello della versione 1, ma possono essere nel mentre stilati i requisiti della versione 2, al tempo t_3, com'è di facile intuzione : Si raccolgono i requisiti per la versione 3, si produce il modello della versione 2, si progetta la versione 1.

1.3 Il linguaggio UML

Il linguaggio UML, acronimo di *Unified Modeling Language*, nasce con l'intento di definire un linguaggio logico-matematico e formale per la progettazione del software. Utilizza dei diagrammi con lo scopo di "sintetizzare" un linguaggio puramente logico.

Verrà utilizzato l'UML per modellare il dominio applicativo ed i dati di interesse, utilizzeremo il cosiddetto **diagramma delle classi e degli oggetti**. Un *oggetto* modella un elemento del dominio di business, la cui esistenza è "autonoma", e può essere identificato appunto come un "oggetto" del mondo reale, identifica una classe, di cui è "estensione", in maniera similare ai linguaggi object-oriented

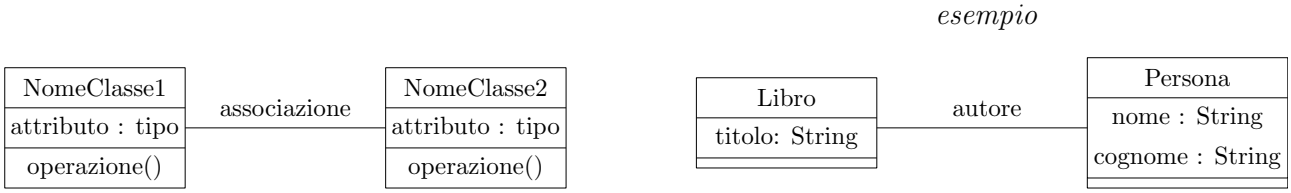
Sarà importante concentrarsi sulle classi piuttosto che sugli oggetti specifici, una classe definisce un nome identificativo, degli attributi e delle operazioni.



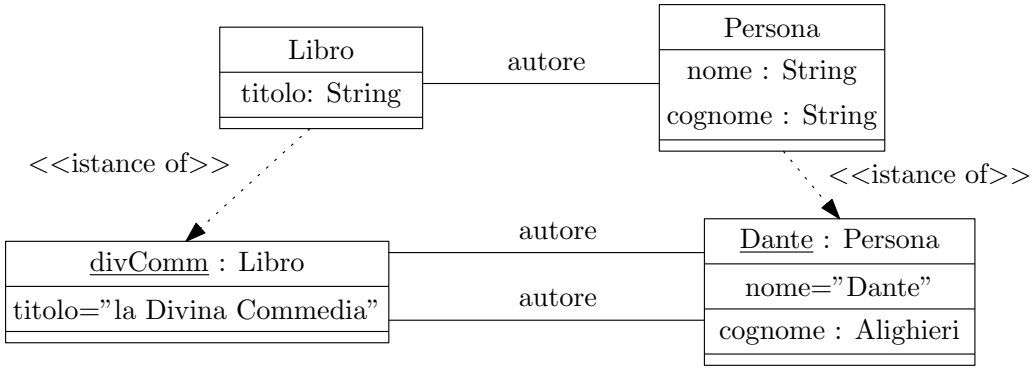
Una classe permette di modellare oggetti dello specifico tipo definito da essa, un oggetto ha un identificatore univoco (sottolineato), possono però esistere due oggetti identici, a patto che differiscano per l'identificatore.

1.4 Associazioni e Link

Un *associazione* definisce un legame fra due oggetti istanza di due classi diverse, si denota con una freccia o linea che collega due classi, e deve presentare un titolo, ad esempio, un oggetto di tipo *Libro*, può essere associato ad un oggetto di tipo *Persona* tramite un'ipotetica associazione *autore*.



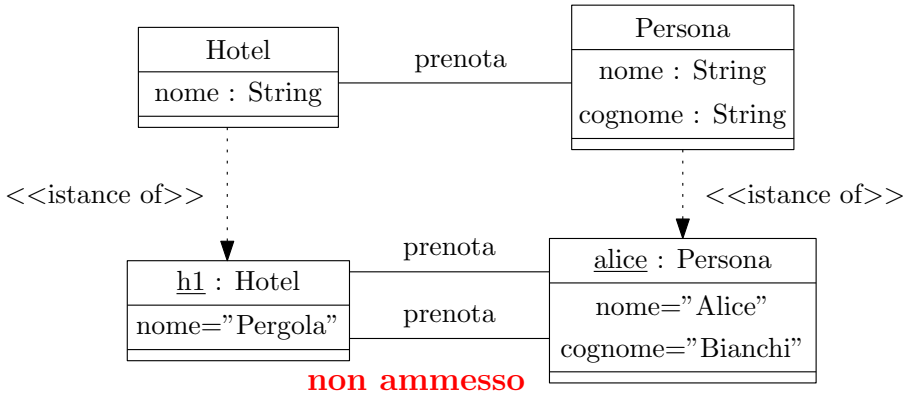
Un *link* non è altro che il corrispettivo delle associazioni, ma sugli oggetti istanza delle classi. Due oggetti identici possono esistere, ma due link identici fra due oggetti no, si immagini l'esempio precedente di autore, non avrebbe senso che una persona sia due volte autore dello stesso libro.



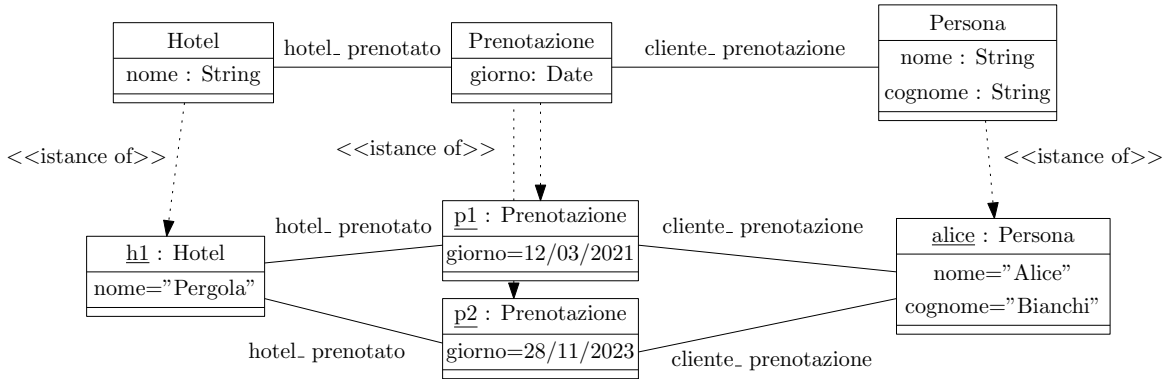
SBAGLIATO : Non ha senso!

1.4.1 Classi Ponte e Molteplicità

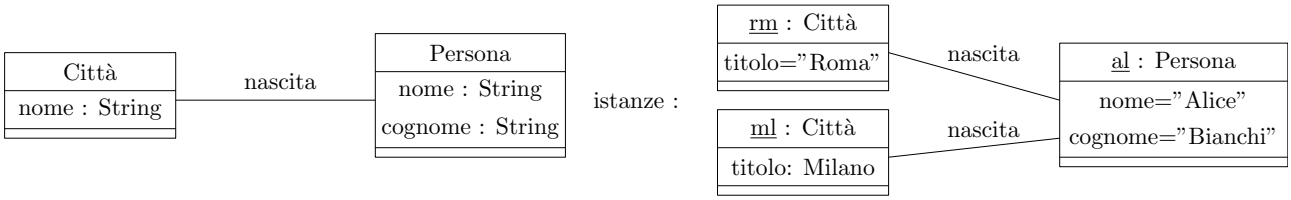
Si consideri adesso il seguente esempio, si vuole progettare un'applicazione che gestire le prenotazioni di un hotel, e si produce il seguente modello UML, con le classi *Hotel* e *Persona* unite dall'associazione "prenota", cosa succederebbe se una persona volesse prenotare 2 volte lo stesso hotel?



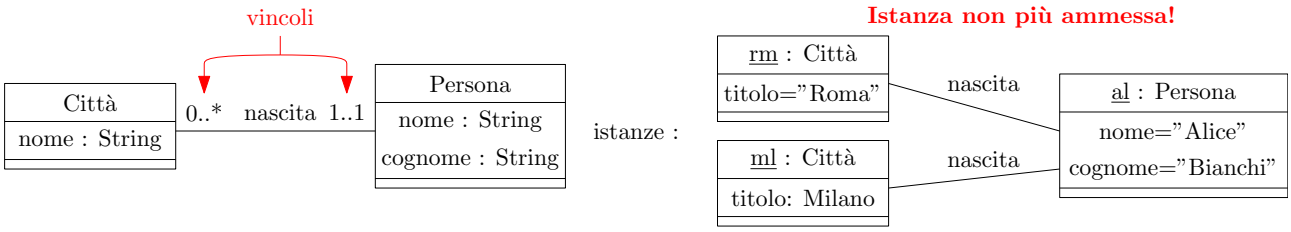
Non è giusto modellare la prenotazione come un associazione, in quanto vogliamo che le prenotazioni esistano come oggetti autonomi, e che uno stesso cliente possa prenotare più volte lo stesso hotel, si necessita di una classe prenotazione che si occupi di tale relazione, una classe di questo tipo è detta **classe ponte**, e nel caso degli hotel, viene implementata nel seguente modo :



Ovviamente, fra le stesse due classi, possono esistere più associazioni diverse, ad esempio, le classi *Libro* e *Persona*, potrebbero essere relazionate da *autore* ed *editore*. Inoltre, un oggetto di una classe C_1, può essere collegato tramite link a due oggetti diversi di una stessa classe C_2, ciò è valido, ma potrebbe causare alcuni errori logici :

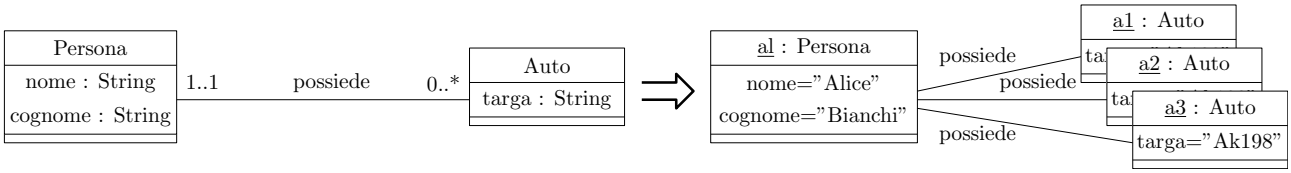


Nonostante lo schema relazionale permetta tali istanze, il fatto che una persona sia nata in due città differenti non rispetta i vincoli del mondo reale, il diagramma è quindi troppo *lasco*, appositamente per situazioni di questo tipo, esistono dei costrutti, detti **vincoli di molteplicità** sulle associazioni, che restringono il possibile numero delle istanze, imponendo delle restrizioni sul numero di link che possono esistere fra due classi.

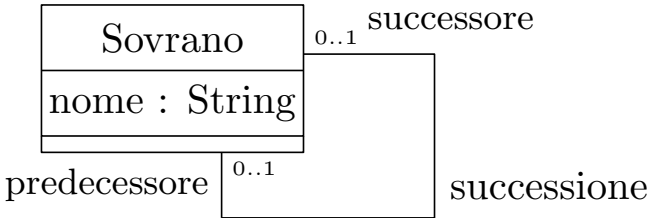


I vincoli di molteplicità vengono aggiunti ai terminali della linea associazione :

- il vincolo **0..*** posto al terminale della classe A, in associazione con la classe B implica che ogni istanza della classe A, dovrà essere coinvolta in un numero di link dell'associazione in questione, che va da 0 ad un qualsiasi numero (ogni istanza di A può essere legata ad un numero qualunque di istanze di B).
- il vincolo **1..1** posto al terminale della classe A, in associazione con la classe B implica che ogni istanza della classe A, dovrà essere coinvolta in un numero di link dell'associazione in questione, che va da 1 ad 1 (ogni istanza di A sarà collegata ad una sola istanza di B).
- (caso generale) il vincolo $k..n$ posto al terminale della classe A, in associazione con la classe B implica che ogni istanza della classe A, dovrà essere coinvolta in un numero di link dell'associazione in questione, che va da k ad n.



Si considerino i seguenti requisiti : Si vogliono rappresentare i sovrani di un regno, di ognuno di loro, è importante considerare il predecessore ed il successore, è possibile in UML creare un link fra due oggetti della stessa classe, con un associazione sulla stessa classe :

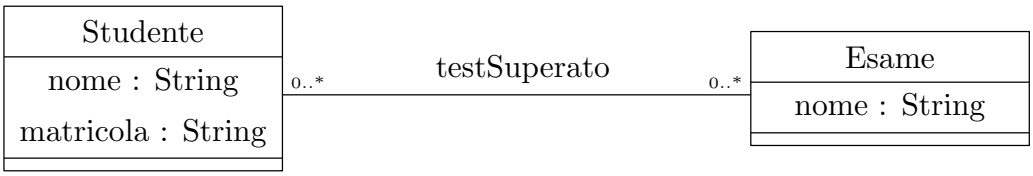


Risulta però *obbligatorio* dare dei nominativi ai **ruoli** posti ai terminali dell'associazione, altrimenti sarebbe impossibile quale delle due classi sta interpretando il ruolo di successore o predecessore. Vorremmo inoltre che ogni sovrano, eccetto il primo e l'ultimo, abbia esattamente un successore ed un predecessore, ma il diagramma in questione permette a qualunque sovrano di violare tali vincoli del mondo reale.

1.4.2 Associazioni con Attributi

Si vuole progettare un sistema che gestisca gli esiti (voti in 30esimi) di più esami sostenuti dagli studenti di un corso di laurea, esisteranno sicuramente le classi *Studente* ed *Esame*.

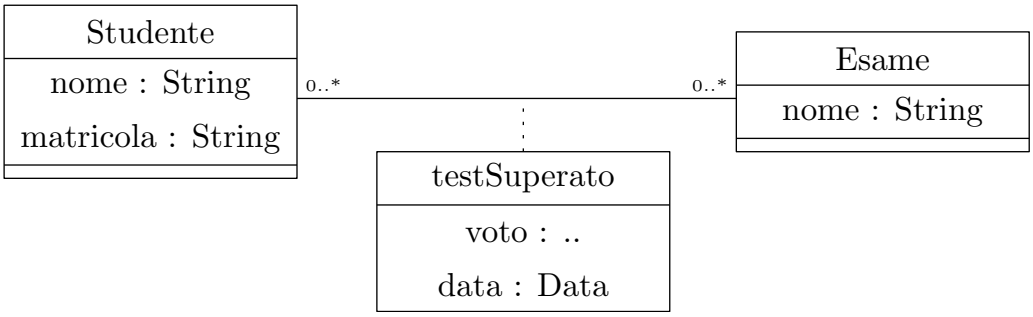
Il problema, è che non è possibile utilizzare una classe ponte, in quanto deve essere impossibile per uno studente, superare lo stesso esame più di una volta. Sarebbe naturale inserire il voto dell'esame in questa ipotetica classe ponte, ma sapendo che non è utilizzabile, dove verrà inserito l'attributo *voto*?



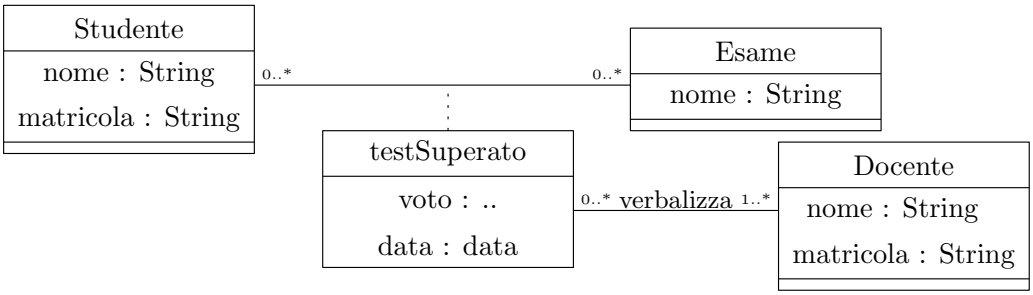
dove inserisco l'attributo *voto* ?

Chiaramente, non posso inserirlo nella classe *Studente*, in quanto ogni studente avrebbe un unico voto per ogni esame, e non posso inserirlo nella classe *Esame*, dato che tutti gli studenti avrebbero lo stesso voto nello stesso esame.

È possibile considerare degli **attributi di associazione**, dando ad ogni link di *testSuperato*, il corrispettivo valore del voto, risulta una soluzione naturale, in quanto il voto è assegnato ad ogni superamento di un test :



Anche se simile, il riquadro *votoSuperato* non rappresenta una classe, in UML è detta *association class*, e anche essa può essere collegata ad altre classi, si supponga ad esempio che vogliamo associare ad ogni test superato, anche il docente che ha verbalizzato il voto, basta collegare l'associazione con attributi ad una classe *Docente* :



1.5 Tipi di Dato

Per ogni attributo di ogni classe abbiamo visto essere necessario considerare il tipo del dato, esiste infatti un insieme di tipi di dato *concettuali*, che siano facilmente implementabili in modo ovvio su qualsiasi sistema o linguaggio di programmazione.

`Intero, Reale, Booleano, Data, Ora, DataOra`

Il fatto è che il linguaggio UML vuole modellare situazioni reali, è quindi necessario considerare dei tipi di dato più restrittivi ed accurati, nell'esempio precedente degli esami, che tipo di dato dovrebbe assumere l'attributo *voto*?

Non possiamo dargli il tipo "intero", in quanto ciò permetterebbe ad un voto di assumere anche

valori come -5 o 25013, e non avrebbe alcun senso per essere un voto di un esame. È possibile considerare dei **vincoli** con un criterio di *specializzazione*, allo scopo di restringere l'insieme dei possibili valori che può assumere un determinato attributo, ad esempio :

Esempi :

budget : Int ≥ 0
attributo1 : Reale $< \pi$

per il voto :

testSuperato
voto : 18..30
data : data

In UML, il tipo $k..n$, indica un *intervallo di numeri interi*, l'attributo in questione può assumere valori da un minimo di k ad un massimo di n, nel caso del voto, assume valori da 18 a 30.

È possibile anche definire esplicitamente l'insieme di valori che possono essere assunti, tramite il tipo *enumerativo* :

Studente
nome : String
matricola : String
genere : {maschio,femmina}

nel campo *genere*, ogni studente potrà assumere il valore "maschio" oppure "femmina"

Se volessimo rappresentare un indirizzo? Si possono creare dei tipi di dato **composti**, costituiti da più tipi di dato, con le eventuali restrizioni (è possibile definire i tipi di dato in un documento separato) :

Studente
nome : String
matricola : String
genere : Mf
indirizzo : Indirizzo

documento separato

Mf : {maschio,femmina}
Indirizzo : {via : String, civico : Int>0, cap : Int>0}

Possono essere definiti anche dei *vincoli di molteplicità* sugli attributi, permettendo ad un oggetto di avere più campi di uno stesso attributo, ad esempio, una classe *Utente* di un social network, può permettere ad ogni utente di avere più indirizzi email, allora l'attributo si definirà nel seguente modo :

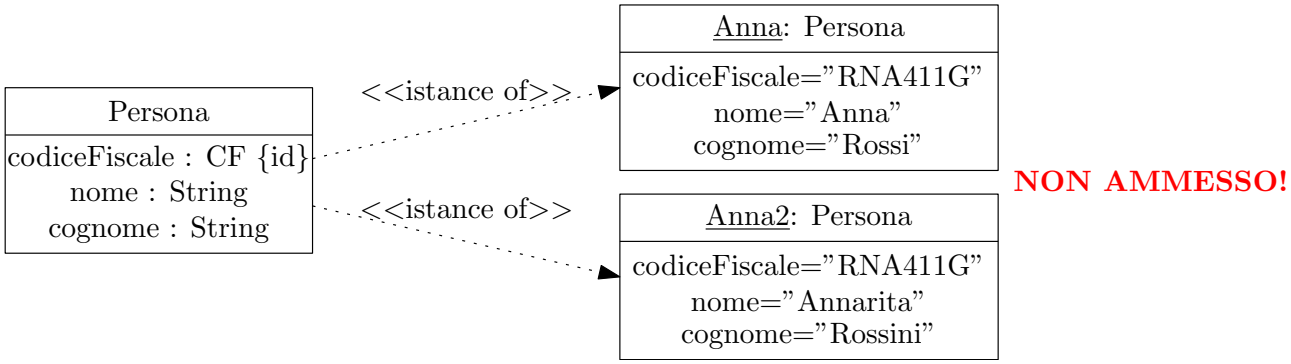
email : String [1..*] *// uno o più indirizzi email*

1.6 Vincoli

Il linguaggio UML permette di aggiungere i cosiddetti **vincoli d'integrità**, delle asserzioni che hanno lo scopo di restringere il possibile insieme delle istanze, ossia degli oggetti ammessi. Una tipologia di vincolo è quella dell'*identificazione di classe*, è un vincolo che, impone a due differenti istanze di una classe, di non poter avere uno o più attributi coincidenti.

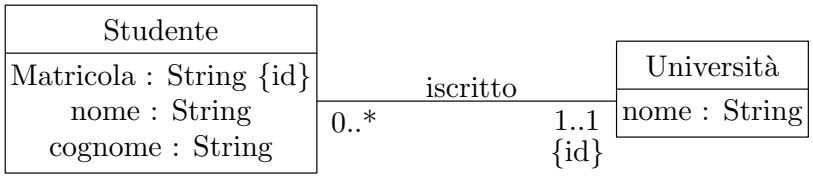
Un tipico esempio può essere fatto per una classe `Persona`, che presenta un attributo `codice fiscale`, è necessario un vincolo di identificazione di classe, che imponga a due differenti istanze di `Persona` di non avere lo stesso `codice fiscale`.

Tale vincolo può essere aggiunto anche su un insieme di attributi, se il vincolo è su `x` ed `y`, due oggetti istanza possono coincidere su `x` ma non su `y`, oppure su `y` ma non su `x`, non possono essere coincidenti su entrambi.



Affianco all'attributo in questione, si inserisce la dicitura `{id}`, possono coesistere anche identificazioni di classe differenti, di solito si usano le diciture `{id1}, {id2}...,{id`k`}` ecc.

Un vincolo di identificazione può anche coinvolgere un'associazione. Se il vincolo è posto su un'associazione da `x` ad `y`, e su un attributo della classe `x`, vuol dire che non possono esistere due istanze di `x` con l'attributo in questione coincidente, che hanno un link verso la medesima istanza di `y` :

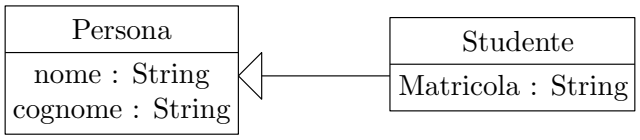


Due studenti con la stessa matricola possono esistere, eslcusivamente se non sono in relazione con la stessa università.

Attenzione : Un vincolo di identificazione di classe può coinvolgere esclusivamente attributi a molteplicità 1..1 e associazioni in cui il ruolo della classe ha molteplicità 1..1.

1.7 Generalizzazione delle Classi

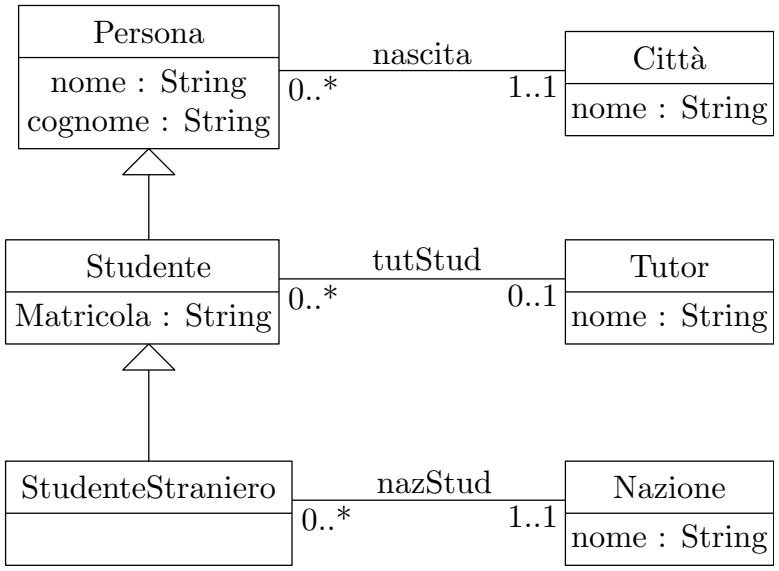
Risultano molto comuni, situazioni in cui diverse classi condividono gli stessi attributi. Il concetto di classe e sotto-classe è ben noto, già dal corso di Metodologie di Programmazione , dove si è affrontata la programmazione orientata agli oggetti, in UML, è possibile definire delle relazioni di classe e sotto-classe, che però risultano ben più "potenti" e flessibili del corrispettivo in Java.



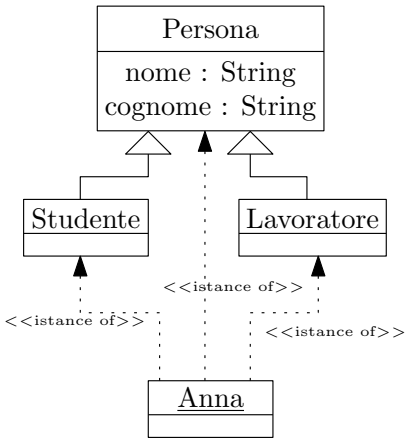
Ogni istanza di *Studente* è anche un istanza di *Persona*, ma ogni istanza di *Persona* non è necessariamente un istanza di *Studente*. Le istanze di *Studente* erediteranno gli attributi della classe *Persona*

Tutti gli attributi, associazioni e le molteplicità della superclasse sono ereditati dalla sottoclasse. Ovviamente la relazione di classe-sottoclasse può essere re-iterata, costruende un "albero" gerarchico, in cui ogni livello eredita gli attributi ed associazioni del livello superiore.

La classe `Studente` sarà sottoclasse di una classe `Persona`, se quest'ultima è in una associazione `nascita` con una classe `Città`, anche `Studente` lo sarà, una sottoclasse di `Studente`, ad esempio, `StudenteStraniero`, erediterà sia le proprietà di `Studente` che quelle di `Persona`.



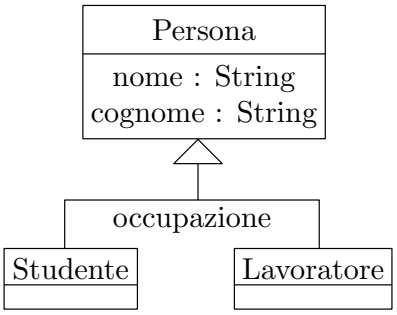
Tutto ciò, aumenta significativamente la complessità dello schema relazionale e del livello degli oggetti, nulla vieta ad un oggetto di appartenere a più classi, nell'esempio precedente, un istanza di `StudenteStraniero`, è anche istanza di `Studente` e `Persona`, però la classe più *specifica* alla quale fa riferimento è appunto `StudenteStraniero`, si definisce per un oggetto quindi l'insieme delle sue classi più specifiche.



L'oggetto Anna è un istanza di Studente, ma anche di Lavoratore, a sua volta, essendo queste ultime sottoclassi di Persona, è anche *implicitamente* istanza di Persona. L'insieme delle sue classi più specifiche è {Studente,Lavoratore}.

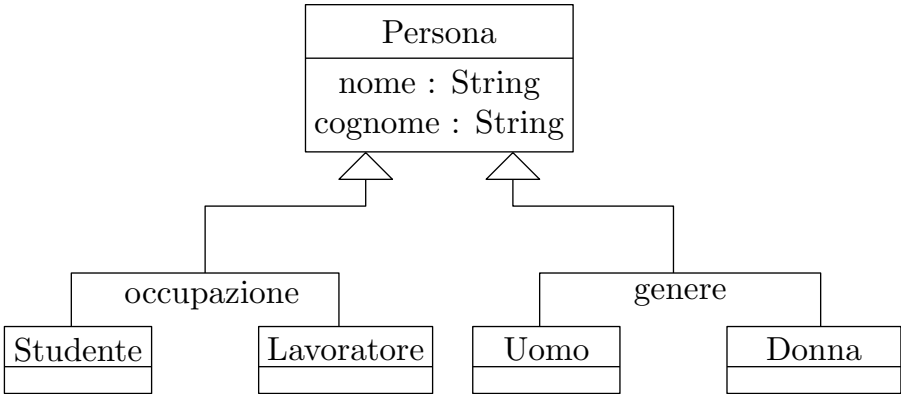
Se diverse classi sono sottoclassi di una classe comune, è possibile utilizzare il costrutto "*is-a*" per denotare tale comportamento del modello. Si ha una classe `a`, e due classi `b` e `c`, entrambe sottoclassi di `a` tramite il costrutto "*is-a*", un oggetto istanza di `a` potrà essere anche :

- Sia istanza di `b` che di `c`.
- Solo istanza di `b`.
- Solo istanza di `c`.
- Ne istanza di `b`, ne di `c`.



è possibile che un oggetto sia Studente, Lavoratore, Studente e Lavoratore oppure nessuno dei due.

In questo caso `Studente` e `Lavoratore` fanno parte della **stessa generalizzazione**, ossia `occupazione`, nulla vieta ad una classe di essere superclasse di generalizzazioni distinte :

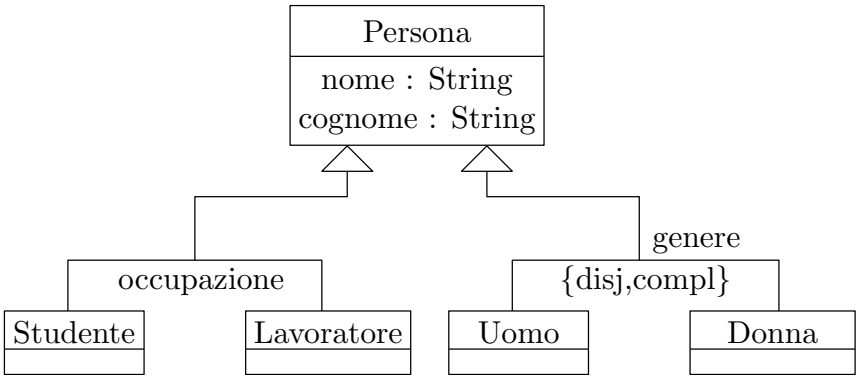


In questo modello, una Persona può essere : Uomo, Donna, Uomo e Donna, ne Uomo ne Donna, e contemporaneamente può essere Studente, Lavoratore, Studente e Lavoratore o nessuno dei due. Risulta ambiguo il fatto che una persona possa essere sia uomo che donna, è necessario imporre allo schema, che ogni persona sia o Uomo o Donna, è possibile considerare dei **criteri sulle generalizzazioni**, che si occupano proprio di gestire queste situazioni.

- **Criterio disjoint** - Impone agli oggetti istanza di una classe soggetta a generalizzazione, di non poter essere istanza di più di una delle sottoclassi in questione. Se nell'esempio precedente, la generalizzazione `genere` avesse il criterio *disjoint*, sarebbe impossibile essere sia Uomo che Donna, ma sarebbe ancora possibile non essere ne Uomo ne Donna.

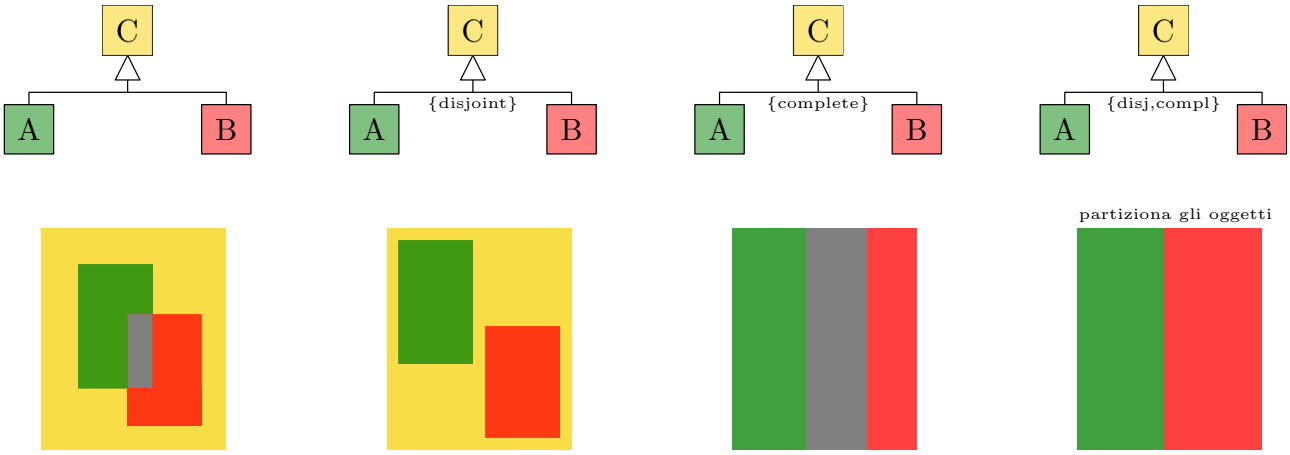
- **Criterio complete** - Impone agli oggetti istanza di una classe soggetta a generalizzazione, di dover essere obbligatoriamente istanza di almeno una delle sottoclassi in questione. Se nell'esempio precedente, la generalizzazione `genere` avesse il criterio *complete*, sarebbe impossibile non essere ne Uomo ne Donna, ma sarebbe ancora possibile non essere sia Uomo che Donna.

A seguito di ciò, è possibile combinare i diversi criteri per poter permettere ad un istanza di persona, di essere necessariamente o Uomo o Donna. Vogliamo quindi che la generalizzazione `genere` consideri entrambi i criteri complete e disjoint, vogliamo invece che la generalizzazione ruolo non abbia criteri, in quanto è possibile che una persona decida di studiare, lavorare, fare entrambi o non fare nulla.



i criteri si scrivono sulla linea della generalizzazione, chiusi fra delle parentesi graffe

Ora le possibili combinazioni di persona sono 8 : Studente Uomo, Lavoratore Uomo, Studente e Lavoratore Uomo, Nullafacente Uomo, Studente Donna , Lavoratore Donna, Studente e Lavoratore Donna e Nullafacente Donna.



UML permette anche ad una classe di ereditare da più classi, anche se tale scelta potrebbe aumentare troppo la complessità del diagramma, e va utilizzata esclusivamente quando necessario, molto spesso è consigliabile procedere diversamente.