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.

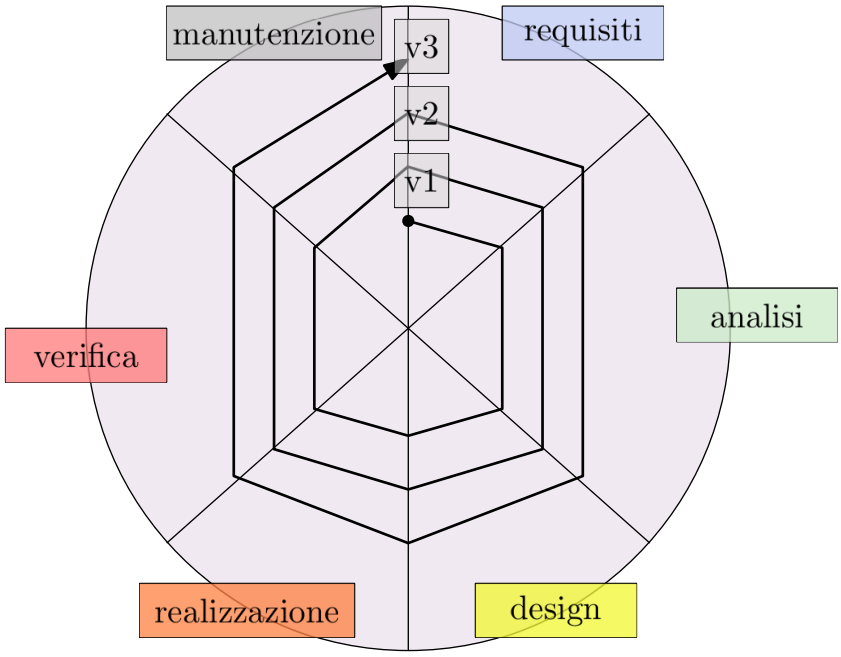
3. **Analisi concettuale dei requisiti** - Sono coinvolti gli analisti, che produrranno uno schema matematico del progetto, dettagliato per filo e per segno, che definirò cosa l'applicazione deve fare indipendentemente dal come. Lo schema prima citato è detto *schema concettuale*, e sarà la base da cui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

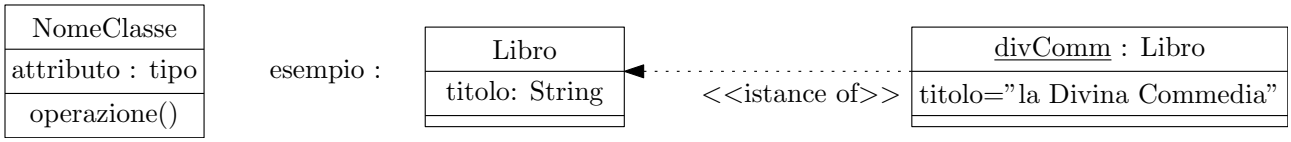
Nulla vieta alle varie fasi di essere eseguite in parallelo, ad esempio, nel tempo t_0 vengono stilati i requisiti per la prima versione del software, nel tempo t_1 gli analisti iniziano a produrre il modello della versione 1, ma possono essere nel mentre stilati i requisiti della versione 2, al tempo 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.

2 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

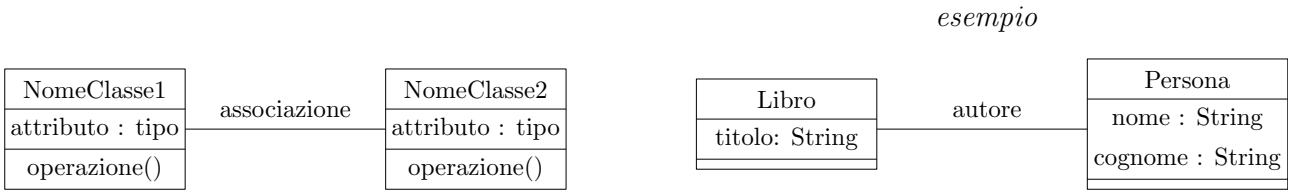
Sarà importante concentrarsi sulle classi piuttosto che sugli oggetti specifici, una classe definisce un nome identificativo, degli attributi e delle operazioni.



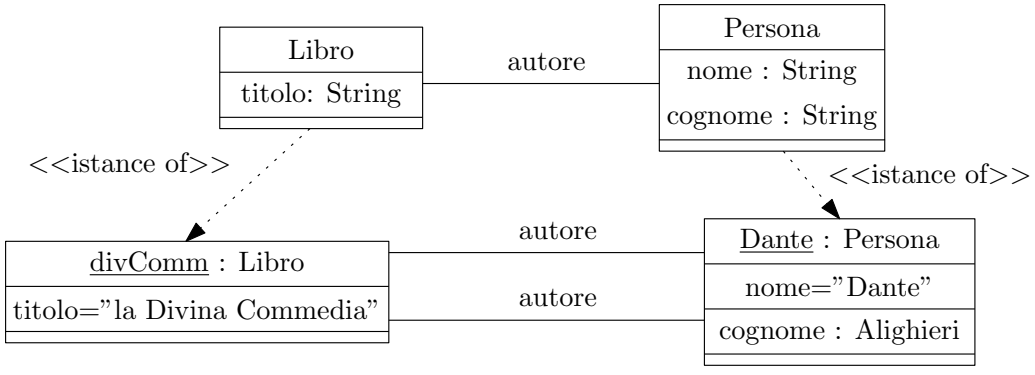
Una classe permette di modellare oggetti dello specifico tipo definito da essa, un oggetto ha un identificatore univoco (sottolineato), possono però esistere due oggetti identici, a patto che differiscano per l'identificatore.

2.1 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*.



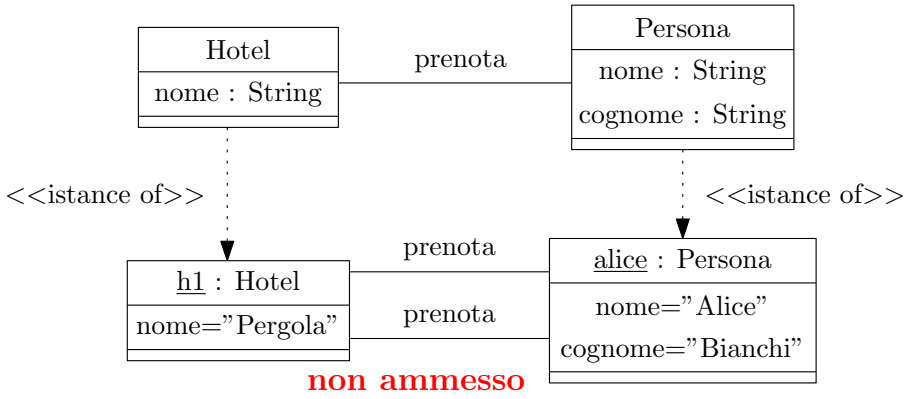
Un *link* non è altro che il corrispettivo delle associazioni, ma sugli oggetti istanza delle classi. Due oggetti identici possono esistere, ma due link identici fra due oggetti no, si immagini l'esempio precedente di autore, non avrebbe senso che una persona sia due volte autore dello stesso libro.



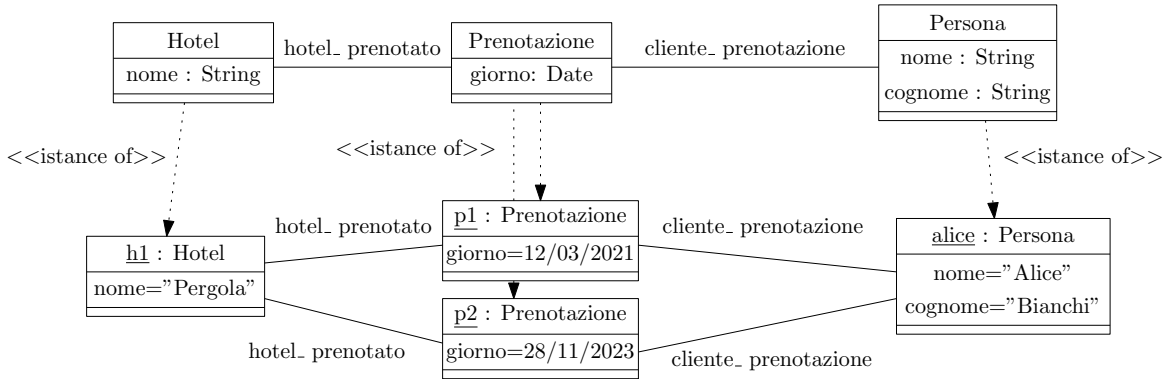
SBAGLIATO : Non ha senso!

2.1.1 Classi Ponte e Molteplicità

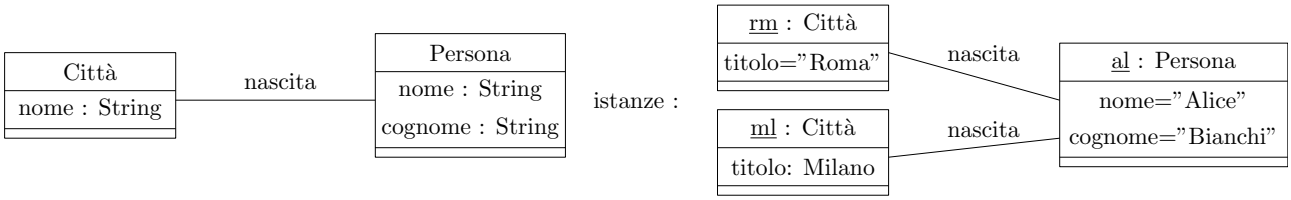
Si consideri adesso il seguente esempio, si vuole progettare un'applicazione che gestire le prenotazioni di un hotel, e si produce il seguente modello UML, con le classi *Hotel* e *Persona* unite dall'associazione "prenota", cosa succederebbe se una persona volesse prenotare 2 volte lo stesso hotel?



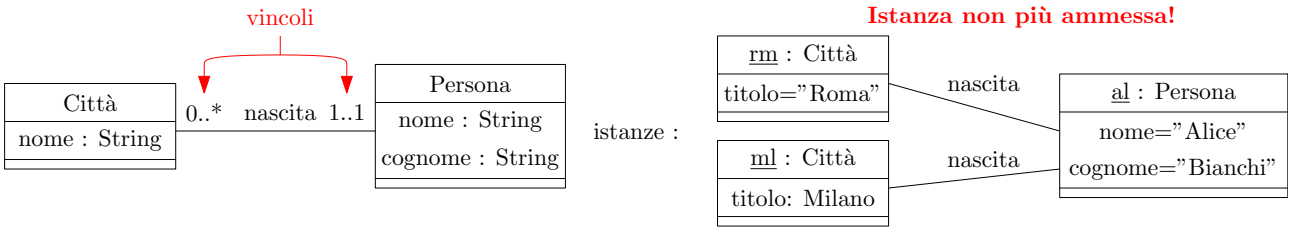
Non è giusto modellare la prenotazione come un associazione, in quanto vogliamo che le prenotazioni esistano come oggetti autonomi, e che uno stesso cliente possa prenotare più volte lo stesso hotel, si necessita di una classe prenotazione che si occupi di tale relazione, una classe di questo tipo è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

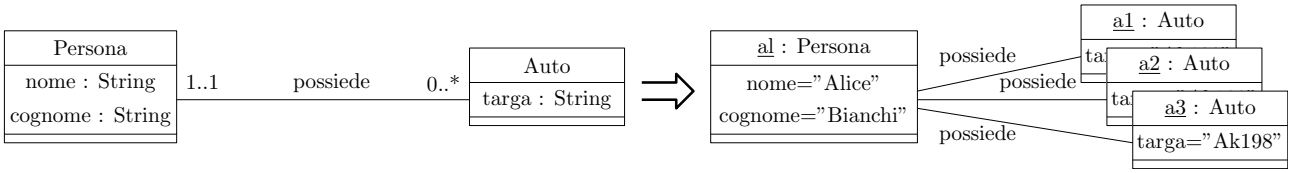


Nonostante lo schema relazionale permetta tali istanze, il fatto che una persona sia nata in due città differenti non rispetta i vincoli del mondo reale, il diagramma è quindi troppo *lasco*, appositamente per situazioni di questo tipo, esistono dei costrutti, detti **vincoli di molteplicità** sulle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

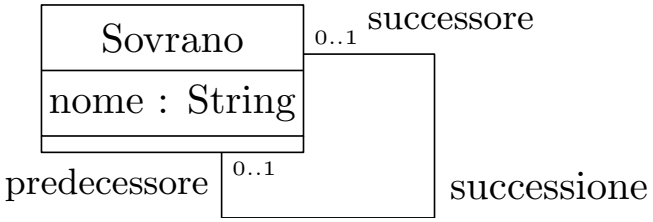


I vincoli di molteplicità vengono aggiunti ai terminali della linea associazione :

- 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 :

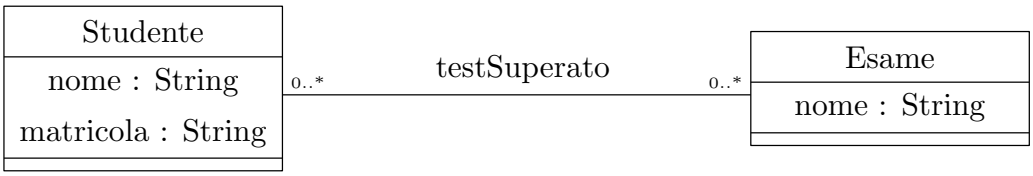


Risulta però *obbligatorio* dare dei nominativi ai **ruoli** posti ai terminali dell'associazione, altrimenti sarebbe impossibile quale delle due classi sta interpretando il ruolo di successore o predecessore. Vorremmo inoltre che ogni sovrano, eccetto il primo e l'ultimo, abbia esattamente un successore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

2.1.2 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*.

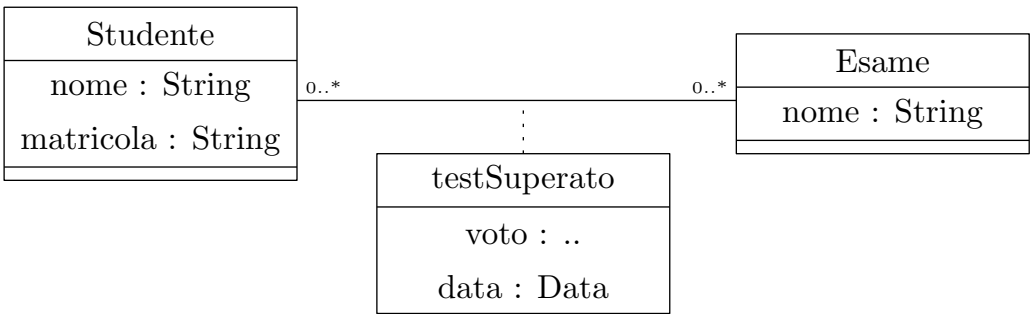
Il problema, è che non è possibile utilizzare una classe ponte, in quanto deve essere impossibile per uno studente, superare lo stesso esame più di una volta. Sarebbe naturale inserire il voto dell'esame in questa ipotetica classe ponte, ma sapendo che non è utilizzabile, dove verrà inserito l'attributo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



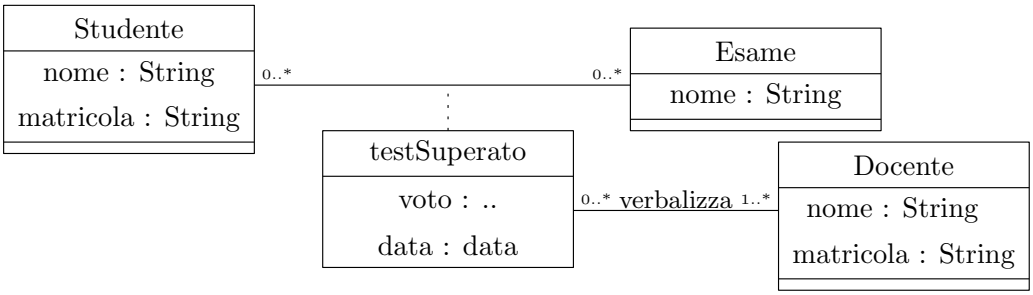
dove inserisco l'attributo *voto* ?

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* :



2.2 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 :

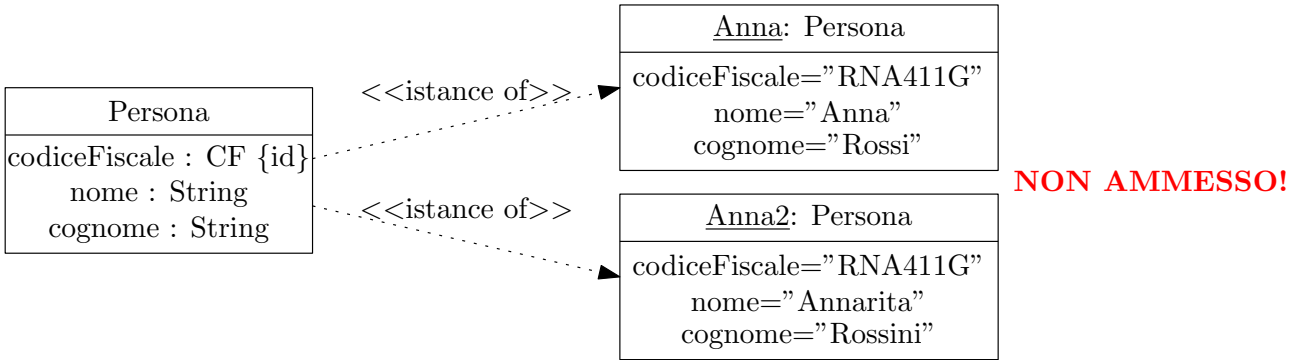
email : String [1..*] *// uno o più indirizzi email*

2.3 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.

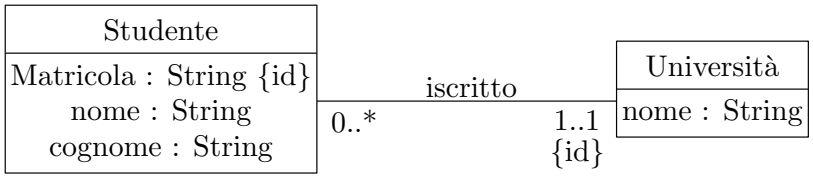
Un tipico esempio può essere fatto per una classe `Persona`, che presenta un attributo `codice fiscale`, è necessario un vincolo di identificazione di classe, che imponga a due differenti istanze di `Persona` di non avere lo stesso `codice fiscale`.

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` :

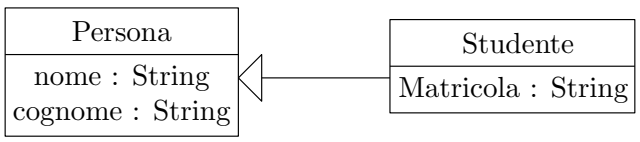


Due studenti con la stessa matricola possono esistere, eslcusivamente se non sono in relazione con la stessa università.

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.

2.4 Generalizzazione delle Classi

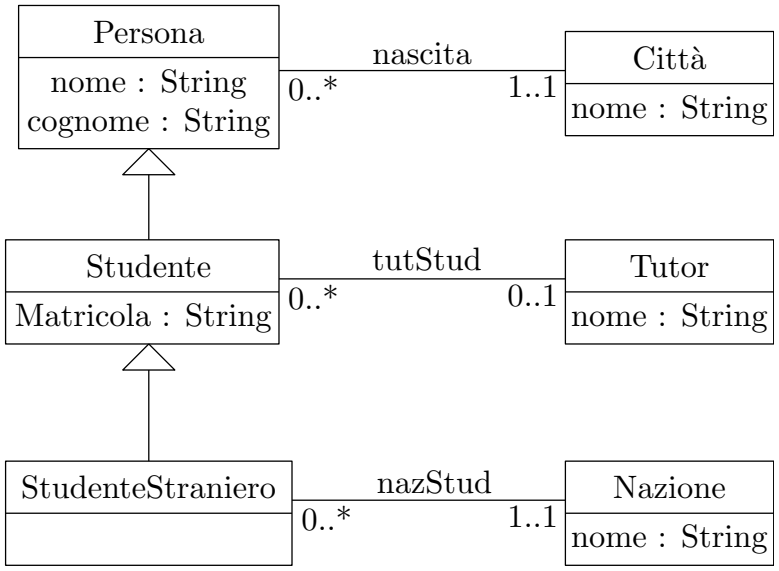
Risultano molto comuni, situazioni in cui diverse classi condividono gli stessi attributi. Il concetto di classe e sotto-classe è ben noto, già dal corso di Metodologie di Programmazione , dove si è affrontata la programmazione orientata agli oggetti, in UML, è possibile definire delle relazioni di classe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



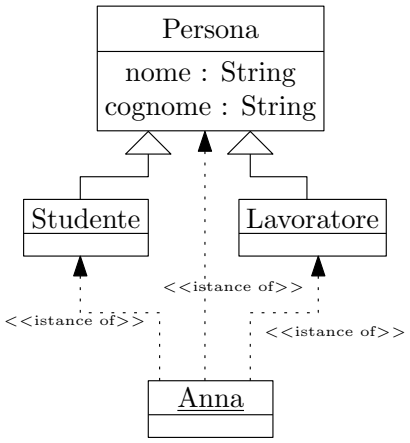
Ogni istanza di *Studente* è anche un istanza di *Persona*, ma ogni istanza di *Persona* non è necessariamente un istanza di *Studente*. Le istanze di *Studente* erediteranno gli attributi della classe *Persona*

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`.



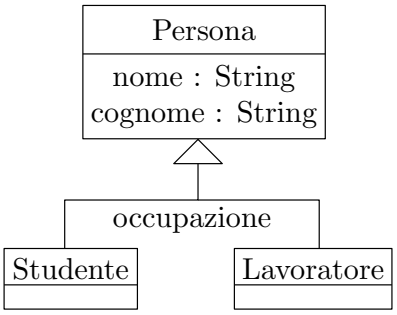
Tutto ciò, aumenta significativamente la complessità dello schema relazionale e del livello degli oggetti, nulla vieta ad un oggetto di appartenere a più classi, nell'esempio precedente, un istanza di `StudenteStraniero`, è anche istanza di `Studente` e `Persona`, però la classe più *specifica* alla 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}.

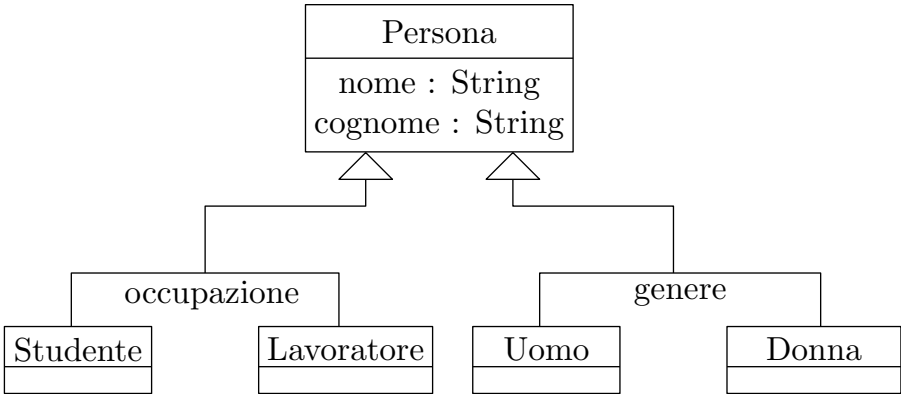
Se diverse classi sono sottoclassi di una classe comune, è possibile utilizzare il costrutto "*is-a*" per denotare tale comportamento del modello. Si ha una classe `a`, e due classi `b` e `c`, entrambe sottoclassi di `a` tramite il costrutto "*is-a*", un oggetto istanza di `a` potrà essere anche :

- 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 :

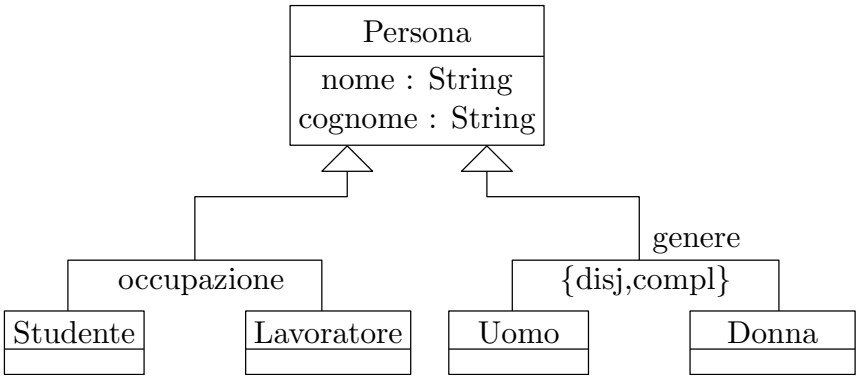


In questo modello, una Persona può essere : Uomo, Donna, Uomo e Donna, ne Uomo ne Donna, e contemporaneamente può essere Studente, Lavoratore, Studente e Lavoratore o nessuno dei due. Risulta ambiguo il fatto che una persona possa essere sia uomo che donna, è necessario imporre allo schema, che ogni 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.

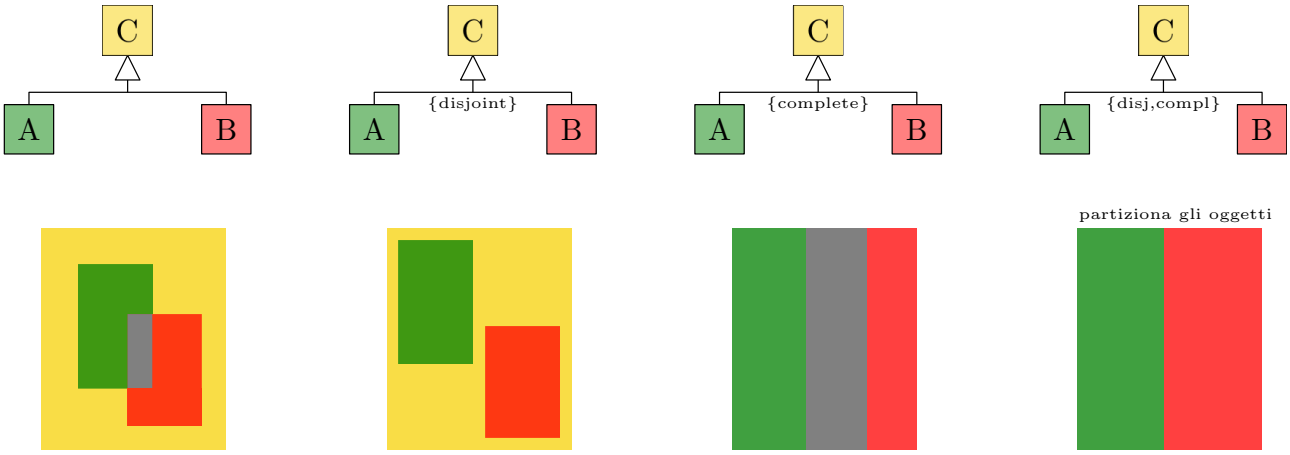
- **Criterio complete** - Impone agli oggetti istanza di una classe soggetta a generalizzazione, di dover essere obbligatoriamente istanza di almeno una delle sottoclassi in questione. Se nell'esempio precedente, la generalizzazione `genere` avesse il criterio *complete*, sarebbe impossibile non essere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

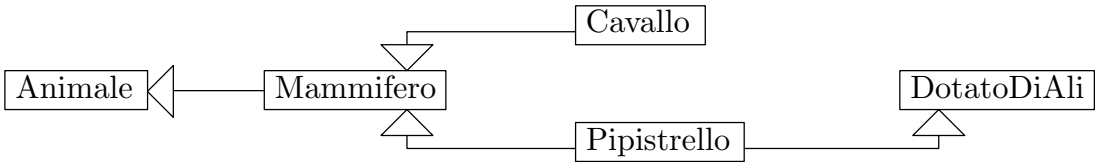


i criteri si scrivono sulla linea della generalizzazione, chiusi fra delle parentesi graffe

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.

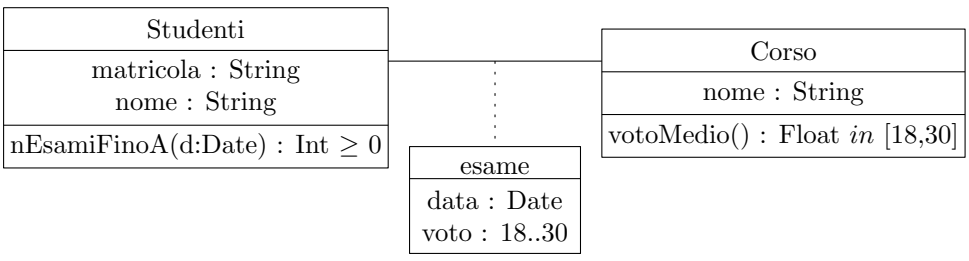


UML permette anche ad una classe di ereditare da più classi, anche se tale scelta potrebbe aumentare troppo la complessità del diagramma, e va utilizzata esclusivamente quando necessario, molto spesso è consigliabile procedere diversamente.



2.5 Operazioni di Classe

Una classe ha un nome, degli attributi e delle *operazioni*, esse definiscono il comportamento della classe e differentemente dagli attributi non sono statici.



Un operazione è una proprietà il cui valore è calcolato a partire dai valori dell'oggetto che la invoca od altri oggetti ad esso correlati, un operazione puà anche cambiare lo *stato* di un oggetto. Quando un operazione modifica un oggetto, si dice che provoca degli *effetti collaterali*.

La sintassi è la seguente, un operazione ha un *nome*, dopo di ché si specificano in delle parentesi tonde i parametri con relativi tipi, alla fine si inserisce il tipo di ritorno dell'operazione. L'operazione definita in una classe x viene sempre invocata da un oggetto istanza di x. L'ereditarietà ovviamente, si applica anche alle operazioni.

in UML quindi vengono definite le operazioni con i loro nomi (evocativi), parametri e tipo di ritorno, ma non viene esplicitato in maniera formale cosa queste operazioni devono calcolare. La descrizione logica-formale di ciò che le operazioni devono fare (non come), viene data in un documento separato.

2.6 Documenti di Specifica

Tali *specifiche* hanno lo scopo di affiancare il diagramma UML, e sono contenute in un documento separato, un tipo di specifiche già presentate sono le specifiche dei **tipi di dato**, viste nel capitolo 2.2, esistono 4 documenti differenti per le specifiche.

2.6.1 Specifica delle Classi

La specifica delle classi è un documento che associa ad ogni classe, appunto, una specifica formale su ciò che rappresenta, di cui per ogni operazione è data la sua descrizione logica-matematica, tale specifica può essere scritta in linguaggio umano (informale) ed in linguaggio puramente logico (formale), per ora vedremo degli esempi di specifiche informali.

La specifica di un operazione segue il seguente template :

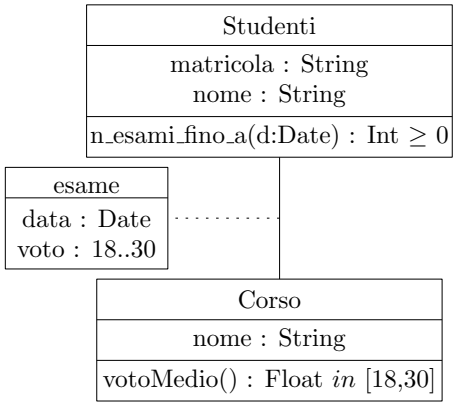
nome_operazione(parametri) : tipo_ritorno

- pre condizioni
  ...
  ...
  ...
- post condizioni
  ...
  ...
  ...

Le specifiche includono alcune parole chiave, come `this` : L'oggetto invocante, oppure `result` : Il valore di ritorno.

Le **post condizioni** danno la definizione matematica dell'operazione e del valore che restituiranno, omettendo eventuali meccanismi di calcolo o implementazioni. Deve essere segnalato se l'operazione modifica o no il livello degli oggetti (le istanze presenti).

Le **pre condizioni** sono le condizioni che devono essere soddisfatte dalle istanze per far si che l'operazione possa essere invocata, definiscono il cosiddetto *contratto* dell'operazione, va anche specificato se l'operazione modifica o no il livello degli oggetti.



Specifica della classe Corso

rappresenta un corso di laurea di cui è possibile sostenere l'esame.

Operazioni :

Operazioni :

votoMedio() : Float *in* [18,30]

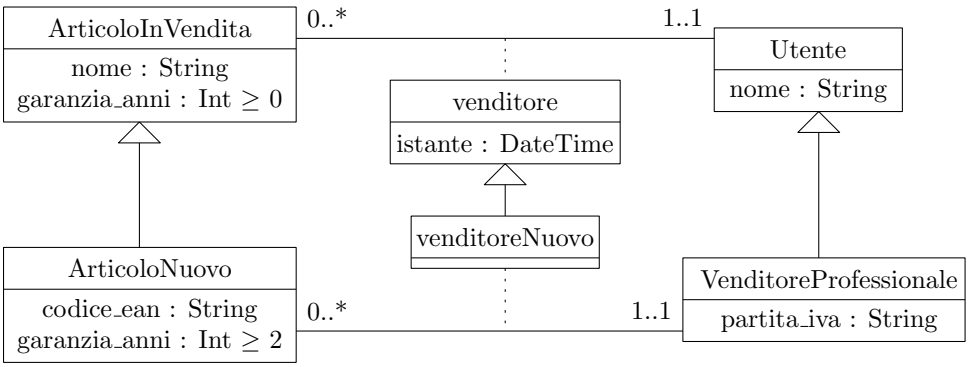
- pre condizioni
  L'oggetto istanza di Corso deve essere coinvolto almeno in un link di tipo esame
- post condizioni
  Sia E l'insieme dei link di tipo esame in cui è coinvolta un istanza di tipo Corso. Sia $e(v)$ l'attributo voto di un elemento di E, sia S la somma di tutti i valori di $e(v)$ per ogni $e \in E$. Sia N la cardinalità di E, il valore dell'operazione è $\frac{S}{N}$.

2.7 Specializzazione

Abbiamo visto come una classe può essere generalizzata, in un rapporto padre-figlio, ossia superclasse-sottoclasse, e che la classe figlia eredita gli attributi e le operazioni della classe padre, il punto è che, essa può anche *specializzare* un attributo desiderato, ossia *restringerne* il dominio, ad esempio:

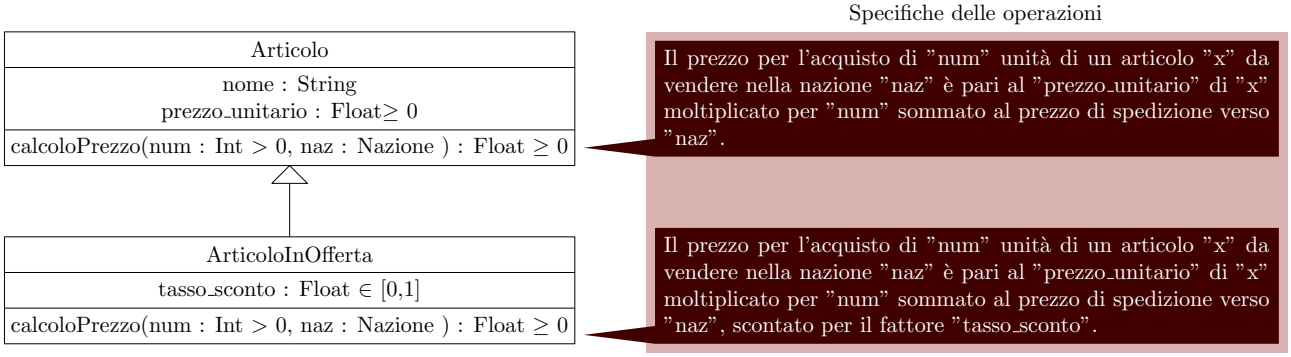
Nel modello sovrastante, un articolo in vendita potrebbe avere una garanzia valida per zero o più anni, ma un articolo nuovo ha una garanzia di minimo due anni, il tipo specializzato deve essere un **sottotipo**, vale a dire che l'insieme dei possibili valori che l'attributo specializzato può assumere, deve essere un sotto-insieme dell'insieme dei valori che può assumere l'attributo che specializza, si dice che ne *restringe* il tipo.

Anche un association class, essendo una classe, può essere specializzata ed essere soggetta a generalizzazioni.



In questo modello, ogni articolo è venduto da esattamente un utente, e gli articoli nuovi possono essere venduti esclusivamente da venditori professionali. Un articolo nuovo è in associazione con un utente, tale associazione però può essere sia *venditore* che *venditoreNuovo*, nel secondo caso, l'utente deve essere necessariamente un venditore professionale.

Supponiamo che vi sia una specializzazione di articolo, ossia un articolo in offerta, il calcolo del suo prezzo deve essere soggetto ad una riduzione, ha senso che vi siano due differenti operazioni *calcoloPrezzo* e *calcoloPrezzoOfferta*? No, dal momento che è possibile *specializzare un operazione*, distinguendo le specifiche, eventualmente anche rendendo completamente differenti le post-condizioni.

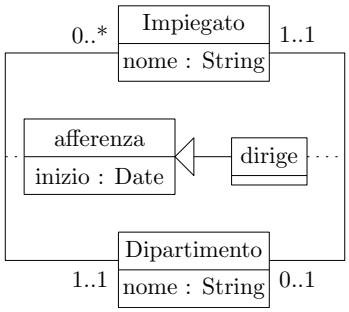


L'importante è che l'operazione specializzata, rispetto a quella che specializza, mantenga:

- Lo stesso numero e tipo di argomenti.
- Lo stesso tipo di ritorno, oppure un sotto-tipo del tipo dell'operazione che spacializza.

2.8 Specifica dei Vincoli Esterni

Ci sono delle regole che il solo diagramma UML non può modellare, si definiscono quindi delle costrizioni *esterne* al diagramma, in un documento separato, che hanno lo scopo di **restringere** il possibile livello degli oggetti, con dei requisiti più **sofisticati**.



[V.Dipartimendo.direttore_anni_afferenza]

Per ogni oggetto dip:Dipartimento, sia dir:Impiegato il direttore di tale dipartimento, tale che (dir,dip):dirige. Deve essere vero che :

$$(dir,dip).inizio \leq adesso - 5\ anni$$

In tale modello, voglio far si che i direttori di un dipartimento, ne afferiscano, da almeno 5 anni, il diagramma UML non permette di modellizzare tale vincolo.

Un vincolo esterno è composto da:

- Un identificatore univoco per riferirsi al vincolo.
- Un'asserzione matematica/logica, un invariante sui dati, ossia delle condizioni che i dati devono necessariamente rispettare.

Per semplicità, inizialmente le asserzioni saranno scritte in linguaggio umano. Se un vincolo coinvolge una sola classe, per mantenere un certo grado di modularità, è buona norma specificare tale vincolo all'interno del documento di specifica della classe in questione.

2.9 Diagramma degli Use-Case

Insieme all'UML, si modella un diagramma che descrive le funzionalità che il sistema deve realizzare, in termini appunti di use-case (scenari di utilizzo), tali use-case possono essere desritti come l'insieme omogeneo di *funzionalità* che un *determinato gruppo* di utenti omogeneo può compiere. Tipicamente coinvolge concetti rappresentati da più classi ed associazioni del diagramma delle classi.

Definiamo come **attore**, il ruolo o la categoria che un utente del sistema (oppure un sistema esterno) può assumere interagendo con il sistema. Un singolo utente può assumere più ruoli, ossia può essere rappresentato da più attori, e diversi utenti possono essere rappresentati dallo stesso ruolo, ossia essere rappresentati dallo stesso attore.

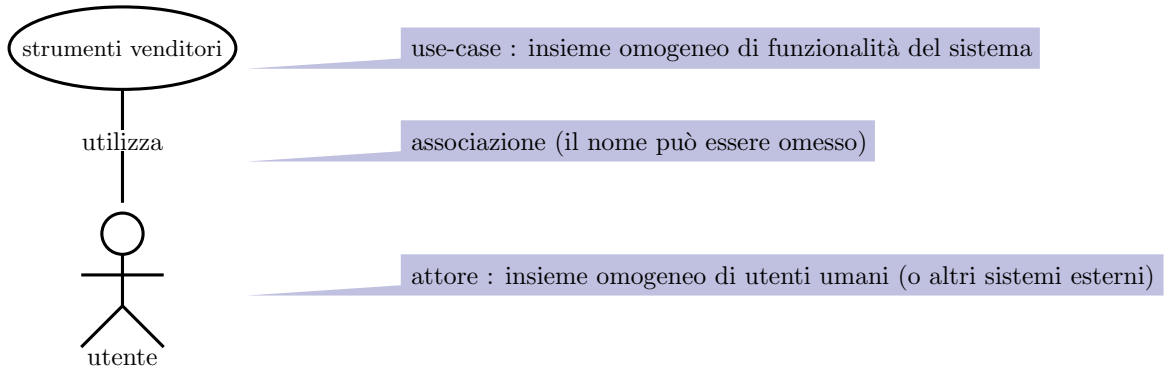
> *Esempio* : Un sistema prevede due attori : Docenti e Studenti. Paolo può essere sia Docente che Studente, mentre sia Alice che Giovanni sono entrambi Studenti.

2.9.1 Semantica del Diagramma

UML fornisce il diagramma degli use-case, è un *grafo*, in cui:

- I **nodi** rappresentano attori e use-case.
- Gli **archi** rappresentano:
  - (*i*) - La possibilità per un attore di invocare uno use-case (a quali funzionalità può accedere un attore).
  - (*ii*) - La possibilità per uno use-case di invocare un altro use-case.
  - (*iii*) - La generalizzazione fra attori e fra use-case (verrà vista in seguito).

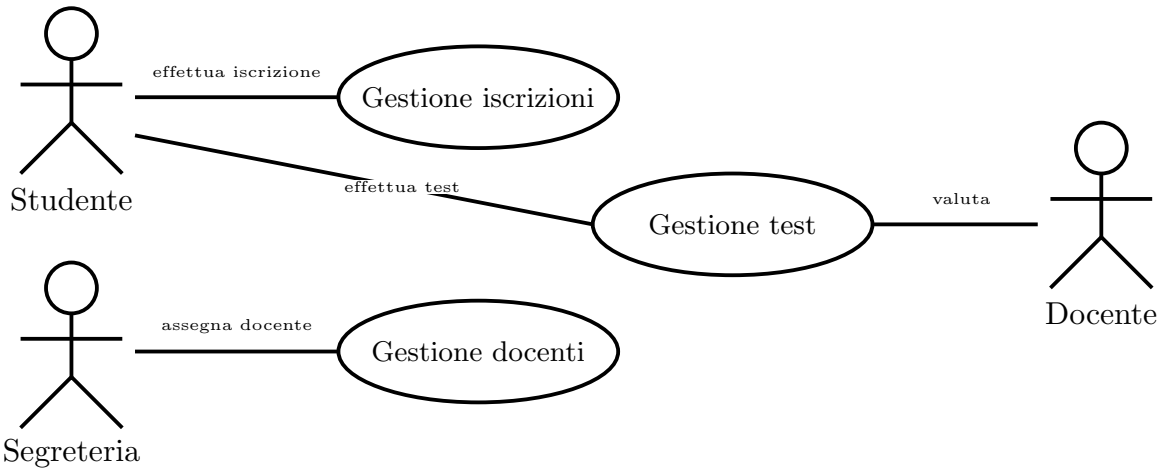
Gli attori sono rappresentati con uno *stickman*, gli use case da un ellisse.



Attenzione : Il fatto che esista un attore *Utente* non implica necessariamente l'esistenza di un una classe Utente nel diagramma delle classi, avremmo la già citata classe Utente *esclusivamente* se il sistema deve rappresentare dei *dati* (attributi) sugli utenti.

Vediamo adesso un *Esempio* di un diagramma degli use-case. Si hanno le seguenti specifiche:

Il sistema deve permettere agli studenti di iscriversi, via web, ai corsi offerti. La segreteria deve poter assegnare i docenti ai singoli corsi. I docenti devono poter inserire i risultati dei test degli studenti: tali test sono somministrati agli studenti utilizzando il sistema.

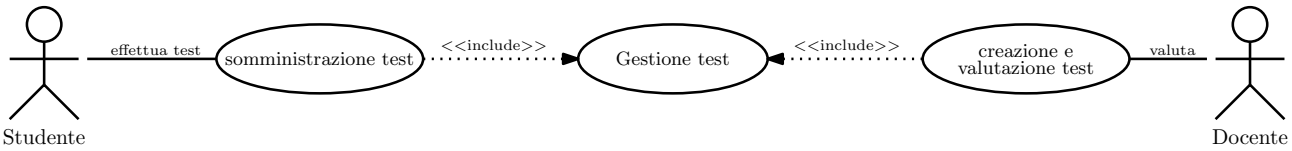


Note sugli use-case:

- *Gestione iscrizioni* : funzionalità che permettono l'iscrizione via web da parte degli studenti.
- *Gestione test* : funzionalità che permettono la creazione, la somministrazione e la valutazione dei test.
- *Gestione docenti* : funzionalità che permettono l'assegnazione dei docenti ai corsi.

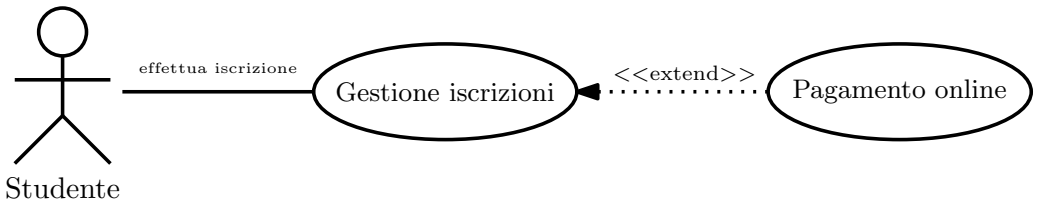
Ogni singolo utente che si interfaccia/interagisce con il sistema modellato deve necessariamente assumere uno fra questi ruoli. Anche se esiste l'attore *Segreteria*, nel diagramma delle classi potrebbe non esistere alcuna classe Segreteria, in quanto di essa non è necessario salvare alcuna informazione.

Nel diagramma sovrastante sorge un problema, sia l'attore *Studente* che *Docente* hanno accesso a tutte le funzionalità dello use-case *Gestione test*, e non solamente a quelle che li riguardano. Nel diagramma, il concetto di **Inclusione**, permette ad uno use-case di utilizzare *alcune* delle funzionalità di un altro use-case.

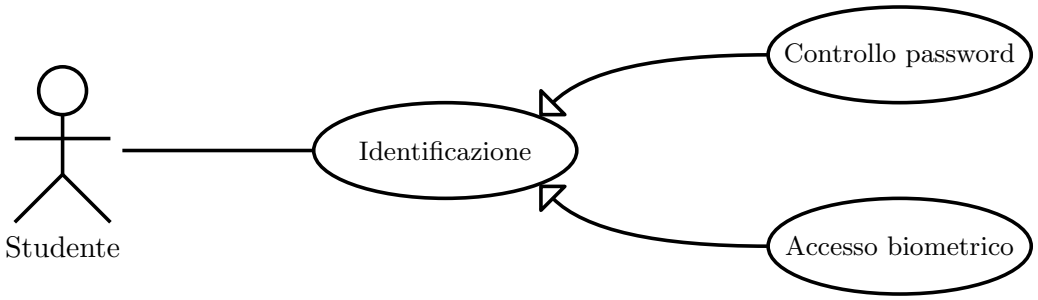


In questo modello, l'attore *Docente* come l'attore *Studente* ha un accesso limitato alle funzionalità dello use-case *Gestione test*, i docenti potranno creare e valutare i test, e gli studenti potranno parteciparvi, *Gestione test* include le funzionalità che permettono la memorizzazione nel sistema dei test e delle risposte degli studenti.

Ci sono delle situazioni in cui uno use-case, eredità le funzionalità di un'altro use-case, aggiungendone di nuove, tale concetto prende il nome di **Estensione**. Riguardo il modello precedente, supponiamo che uno studente che si iscrive ad un test, ha anche la possibilità, se volente, di pagare online, lo use-case *Iscrizione* verrà quindi esteso. Il pagamento online è un caso particolare di iscrizione.

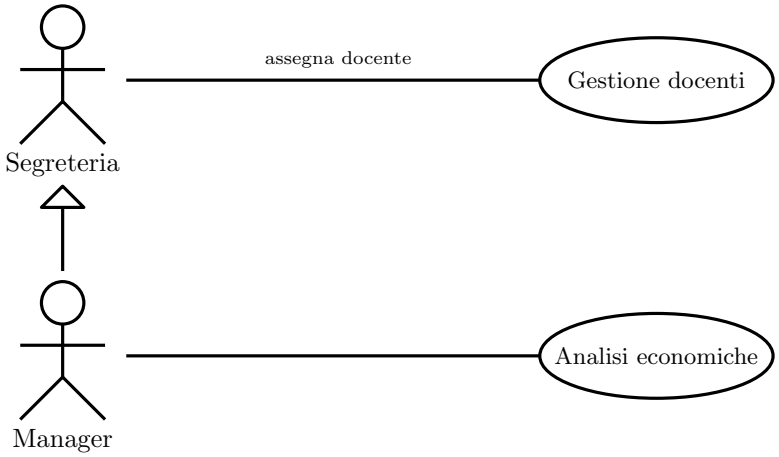


Alcuno funzionalità di uno use-case, in alcuni casi particolari, possono essere *rimpiazzate* con le funzionalità di un'altro use-case, è possibilie **generalizzare uno use-case**. Supponiamo che gli studenti devono potersi identificare, e tale identificazione può avvenire tramite una password, oppure tramite un accesso biometrico (impronta digitale o iride):



Nota : nei diagrammi degli use-case non possiamo usare generalizzazioni uniche che coinvolgono più sotto-usecase, né tantomeno vincoli {disjoint} e {complete}.

Un'altra possibilità, è quella di far si che un attore "erediti" tutte le associazioni di un altro attore, potendo invocare i suoi use-case, è possibile **generalizzare un attore**. Supponiamo che in un sistema, i manager possano fare le veci della segreteria ed accedere a gli use-case ad essa dedicati:



In questo modello un manager eredita le funzionalità della segreteria, e può anche esso gestire i corsi dei docenti.

Attenzione : Il diagramma non implica che esistano le classi Segreteria e Manager nel diagramma delle classi, né tantomeno che la classe Manager sia una sottoclasse di Segreteria.

2.9.2 Specifiche degli Use-Case

Il diagramma UML degli use-case è semplice e poco elaborato, più semplice del diagramma delle classi in termini di quantità dei costrutti, da una visione di alto livello riguardo:

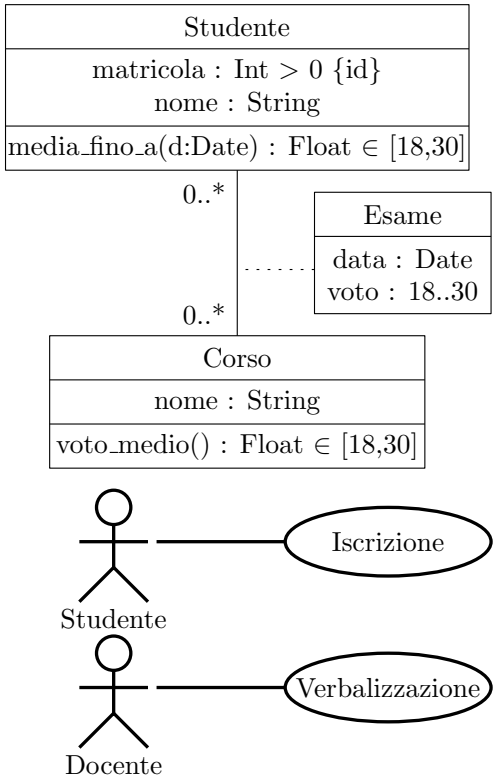
- Quali sono gli attori che possono interagire con il sistema.
- A quali macro-funzionalità gli attori definiti possono accedere.

Definiscono come tali macro-funzionalità sono organizzate e modularizzate, deve essere un diagramma comprensibile per il committente, e *non definisce* le singole operazioni all'interno di ogni use-case, de facto, ogni singolo use-case deve essere affiancato da un apposito *documento di specifica* dettagliato.

Le operazioni degli use-case, non presentano alcun oggetto di invocazione, possono come le operazioni di classe, non avere alcun tipo di ritorno, si guardi il seguente esempio:

Specifica dello use-case Iscrizione
iscrizione (mat : Int>0, nome : String) : Studente
pre-condizioni:
- non esiste alcun oggetto di classe Studente con valore "mat" per l'attributo matricola.
post-condizioni:
-viene creato e restituito un nuovo oggetto result:Studente con i valori "mat" e "nome" per, rispettivamente, gli attributi matricola e nome.
...
(specifica delle altre operazioni dello use-case)

Specifica dello use-case Verbalizzazione
verbalizza (s : Studente, c : Corso, d : Date, v : 18..30)
pre-condizioni:
- Non esiste già un link (s,c) dell'associazione "esame" .
post-condizioni:
- viene creato il link (s,c) di assoc. "esame", con valori "d" e "v" per gli attributi "data" e "voto".
...
(specifica delle altre operazioni dello use-case)



Si noti come nel diagramma delle classi non esiste alcuna classe Docente nonostante esista il medesimo attore, questo perchè il sistema non richiede di memorizzare le informazioni relative ai docenti.

3 Logica di Primo Ordine

Una logica, è una famiglia di linguaggi formali, che ha lo scopo di *rappresentare* l'informazione e *manipolare* la conoscenza. Ogni logica è fornita di una **sintassi** e di una **semantica**, la sintassi definisce una serie di simboli e la struttura che le formule devono assumere per essere considerate valide, la semantica definisce il significato di tali formule.

Nelle logiche classiche, ogni formula, in base al "mondo" sulla quale è applicata, può essere vera oppure false. Per definire la sintassi devo stabilire:

- Quali simboli posso utilizzare (l'alfabeto)
- Quali sequenze finite di simboli possono comporre una formula

La semantica invece stabilisce la *verità* delle formule nei cosiddetti "mondi" possibili, ossia le **interpretazioni**. Nella logica di primo ordine, detta anche **FOL** si stabilisce

- Sintassi
- Semantica
  - Interpretazione
  - Assegnamento delle variabili
  - Modello
  - Valutazione di una formula
- Soddisfacibilità
- Insoddisfacibilità
- Validità

3.1 Sintassi della FOL

La logica di primo ordine segue la seguente sintassi, vi sono :

- un insieme $\mathcal{V}$ di variabili
- un insieme $\mathcal{F}$ di simboli di funzione, dove ognuno di essi ha associata la sua arità
- un insieme $\mathcal{P}$ di simboli di predicato, dove ognuno di essi ha associata la sua arità
- dei connettivi logici, ossia : $\neg, \wedge, \vee, \rightarrow, \leftrightarrow$
- dei quantificatori, universale $\forall$ ed esistenziale $\exists$
- altri simboli speciali come la virgola

L'arità dei simboli di funzione e dei simboli di predicato è il numero di parametri/argomenti che accettano, un simbolo di funzione f di arità 1 si denota $f/1$. Ad esempio, definisco il simbolo di funzione $succ/1$, che associa ad ogni numero intero il suo successivo, se $x \in \mathbb{N}$, $succ(x) = x + 1$.

I simboli di funzione servono ad identificare degli *oggetti* del mondo, dati degli argomenti, che sono oggetti del mondo, restituiscono un altro oggetto del mondo, i simboli di funzioni di arità 0 sono detti simboli di costante. I simboli di predicato invece, definiscono delle *proprietà*, dati degli argomenti, un simbolo di predicato può essere vero o falso, ad esempio, per il simbolo $doppio/2$, risulta che $doppio(x, y) = True \iff x = 2 \cdot y$.

Un altro esempio può essere un simbolo di predicato $uomo/1$ che è vero se e solo se l'argomento passato è un uomo, ma chi decide se un oggetto del mondo è un uomo o no? E chi decide qual'è la funzione effettiva associata ad un simbolo di funzione? Ciò verrà visto in seguito nel capitolo riguardante l'interpretazione.

Una formula nella FOL si divide in due piani, il linguaggio dei *termini* ed il linguaggio delle *formule*.

Definizione di termine

- Ogni variabile è un termine
- Ogni simbolo di funzione di arità 0 è un termine
- Se f/n è un simbolo di funzione, dove $n \geq 1$, e $t_1, t_2 \ldots, t_n$ sono dei termini, allora anche $f(t_1, t_2 \ldots, t_n)$ è un termine.

Consideriamo il seguente esempio

$$\mathcal{F} = \{zero/0, succ/1, socrate/0, padre/1\} \qquad \mathcal{V} = \{MiaVar, X\}$$

I seguenti sono dei termini

$$\begin{array}{cc} zero & padre(padre(socrate)) \\ MiaVar & padre(succ(X)) \\ succ(zero) & succ(succ(zero)) \end{array}$$

Definizione di formula

- Se p/n è un simbolo di predicato e $t_1, t_2 \ldots, t_n$ sono dei termini, allora $p(t_1, t_2 \ldots, t_n)$ è una formula (atomica).
- Se ϕ e φ sono due formule, allora anche le seguenti sono formule
  - (ϕ) $\quad \phi \vee \varphi$ $\quad \phi \rightarrow \varphi$
  - $\neg\phi$ $\quad \varphi \wedge \phi$ $\quad \phi \leftrightarrow \varphi$
- Se ϕ è una formula e X è una variabile, allora anche le seguenti sono formule

$$\forall X\phi \quad \exists X\phi$$

Esiste un simbolo di predicato speciale, utilizzabile in ogni formula, ossia $= /2$, se x ed y sono identici, ossia sono lo stesso oggetto del mondo, allora $= (x, y)$ sarà vera, per comodità, tale simbolo verrà utilizzato scrivendo $x = y$ piuttosto che $= (x, y)$. Analogamente, verrà utilizzato $x \neq y$ piuttosto che $\neg = (x, y)$.

Consideriamo il seguente esempio

$\mathcal{F} = \{zero/0, succ/1, socrate/0, padre/1\}$ $\qquad$ $\mathcal{P} = \{doppio/2, somma/3, uomo/1, mortale/1\}$

Le seguenti sequenze di simboli sono formule

$doppio(succ(succ(zero)), X)$ $\quad$ $\exists X doppio(succ(succ(zero)), X)$ $\quad$ $\forall X doppio(succ(succ(zero)), X)$

$somma(succ(zero), zero, succ(zero))$ $\quad$ $\forall X \forall Y somma(X, X, Y)$ß$doppio(X, Y)$ $\quad$ $mortale(socrate)$

$(\forall X \exists Y doppio(X, Y)) \wedge (\forall I \forall J \exists K somma(I, J, K))$ $\quad$ $mortale(socrate) \wedge mortale(padre(socrate))$

3.2 Semantica della FOL

La verità di una formula e dei predicati dipendono dal mondo in cui sono espressi, nella logica proposizionale:

$$(A \vee \neg B) \wedge C \begin{cases} \text{è vera se } A = 1,\ B = 1,\ C = 1 \\ \text{è false se } C = 0 \end{cases}$$

Bisogna definire la nozione di interpretazione, essa fornisce i valori di base per i predicati e per i termini. Tutte le formule atomiche sono le cosiddette *lettere proposizionali*, l'interpretazione è una funzione I che associa ad ogni lettera proposizionale un valore di verità, la pre-interpretazione riguarda l'interpretazione dei termini.

Consideriamo i seguenti simboli

$$\mathcal{F} = \{socrate/0, padre/1\} \quad \mathcal{P} = \{Uomo/1, Mortale/0\}$$

Per valutare la formula

$$(\forall X Uomo(X) \rightarrow Mortale(padre(X))) \wedge Uomo(socrate)$$

Bisogna fornire un insieme D di oggetti del dominio, una corrispondenza fra i simboli di funzione e funzioni effettive su D, ed una corrispondenza fra le variabili e gli elementi di D.

Le funzioni nella FOL sono *totali*, ogni singolo elemento del dominio D può essere valutato da una funzione associata al relativo simbolo di funzione, data un'interpretazione, una volta risolti tutti i termini, è possibile valutare qualsiasi formula complessa.

I simboli di predicato sono definiscono delle *relazioni matematiche* su D^n dove n è l'arità del simbolo in questione. Un simbolo di predicato risulta vero se gli oggetti del dominio coinvolti sono contenuti nella relazione associata al simbolo, si consideri il seguente esempio

$$D = \{a, b, c\}$$

$$\mathcal{P} = \{p/2\} \quad \text{la relazione associata a } p \text{ è } p = \{(a, b), (b, c)\}$$

$$p(a, b) = True \qquad p(a, c) = False \qquad p(b, c) = True$$

Vediamo adesso um *esempio* esaustivo, si considerino i seguenti simboli:

$$\mathcal{F} = \{pippo/0, pluto/2, ciro/0\} \qquad \mathcal{P} = \{Paperino/2, Ugo/0\} \qquad \mathcal{V} = \{X, Y\}$$

Considero la formula sintatticamente corretta:

$$Paperino(pippo, pluto(X, pippo)) \rightarrow \neg(Ugo \vee Paperino(pippo, ciro))$$

Per valutarla, necessito di un intepretazione I, considero il seguente dominio

$$D = \{a, b, c, d\}$$

Definisco ora le funzioni associate all'interpretazione I

$$I(pippo) : D^0 \rightarrow D \text{ tale che } pippo() = d$$

$$I(ciro) : D^0 \rightarrow D \text{ tale che } ciro() = a$$

$$I(pluto) : D^2 \to D \text{ tale che}$$

$$\begin{array}{ll} pluto(a,a) = a & pluto(b,c) = a \\ pluto(a,b) = a & pluto(b,d) = a \\ pluto(a,c) = a & pluto(c,c) = d \\ pluto(a,d) = b & pluto(c,d) = d \\ pluto(b,b) = c & pluto(d,d) = a \end{array}$$

Assegno le variabili

$$I(X) = c \qquad I(Y) = c$$

A questo punto, definisco le relazioni associate ai simboli di predicato

$$Paperino = \{(a,b),(b,b),(d,d)\} \text{ è una relazione su } D^2$$

Ugo è una relazione su D^0, questi simboli di predicato hanno un valore fisso, sono o veri o falsi, in questo caso, definisco $Ugo = True$. A questo punto posso interpretare la formula

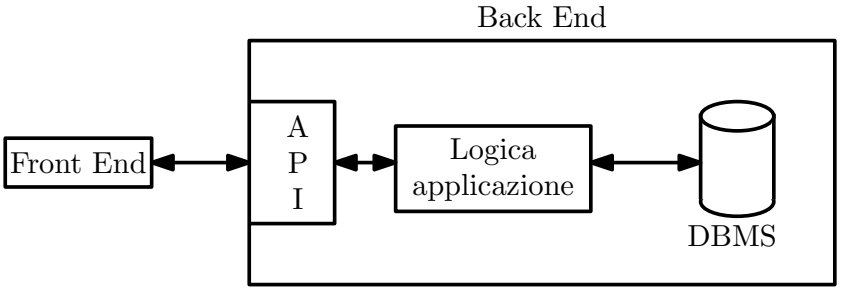
$Paperino(pippo, pluto(X, pippo)) \to \neg(Ugo \vee Paperino(pippo, ciro))$
$Paperino(d,d) \to \neg(Ugo \vee Paperino(d,a))$
$True \to \neg(True \vee False)$
$True \to \neg(True)$
$True \to False$
$False$

3.3 Valutazione dei Termini

4 Progettazione di Basi di Dati

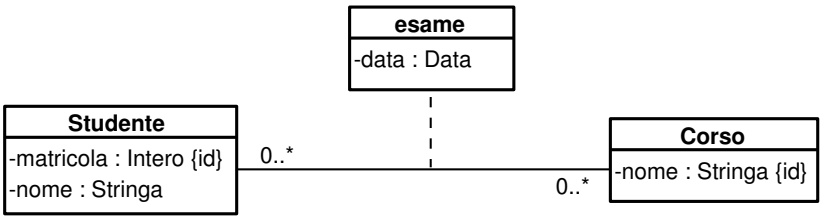
Una volta aver definito lo schema concettuale del sistema da realizzare, è possibile derivarne una base di dati in pochi passaggi, tramite una metodologia precisa e delineata. Il processo di "trasformazione" dal diagramma UML (con relativi vincoli) ad una base di dati, è suddiviso in tre passaggi

1. Far corrispondere ogni tipo di dato concettuale ad un tipo di dato implementato nel DBMS.
2. Progettazione dello schema relazionale, tenendo conto dei vincoli.
3. Progettazione delle specifiche realizzative inerenti alle operazioni di UseCase e classi.



Quindi, dato in input un diagramma UML, si vuole definire una base di dati, si osservi il seguente esempio:

input :



output :

```
tabella Studente(matricola :  integer, nome :  varchar)
tabella Corso(nome :  varchar)
tabella esame(studente :  integer, corso :  varchar, data :  Data)
    foreign key studente references Studente(matricola)
    foreign key corso references Corso(nome)
```

Scrivere manualmente la base di dati a partire dal diagramma UML comporta un rischio di errore maggiore, verrà utilizzata una metodologia **scalabile e robusta**, suddivisa in due fasi:

1. **Ristrutturazione**
2. **Traduzione diretta**

4.1 Ristrutturazione del Diagramma delle Classi

La ristrutturazione, consiste nel definire un nuovo diagramma delle classi, adatto all'implementazione su una base di dati, tale diagramma, rispetterà le seguenti proprietà

- Definisce solamente classi le cui istanze saranno memorizzate sulla base di dati
- Non comprende operazioni di classe
- Risulta *equivalente* all'originale, ossia, rappresenta gli stessi livelli estensionali del diagramma di partenza, anche se diverso nella struttura
- Contiene i costrutti più semplici del UML, ovvero
  - classe
  - associazione
  - attributi i cui tipi di dato, sono anche disponibili nel linguaggio `SQL`
  - molteplicità degli attributi, al massimo pari ad 1

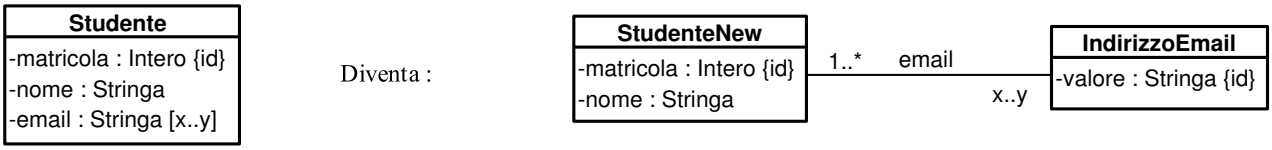
La ristrutturazione, è composta da 6 passi fondamentali:

1. Eliminazione di attributi multivalore
2. Sostituzione dei tipi di dato
3. Eliminazione delle generalizzazioni tra classi ed associazioni
4. Definizione degli identificatori per ogni classe
5. Selezione di un identificatore primario per ogni classe

4.1.1 Procedimento della Ristrutturazione

PASSO 1

Il primo passo consiste nel rimuovere dallo schema tutti quegli attributi, la cui molteplicità risulta superiore ad 1, in quanto un database relazionale non può gestire attributi multivalore. Gli attributi multi valore, diventeranno una classe separata, la cui classe originale proprietaria sarà collegata tramite un associazione, di molteplicità uguale a quella dell'attributo ormai eliminato.



Si abusa quindi del concetto di classe per adattare il diagramma alla tecnologia delle basi di dati relazionali.

PASSO 2

A questo punto, si vogliono eliminare tutti i tipi di dato concettuali, per sostituirli con dei tipi di dato esistenti ed implementati nel DBMS, per i tipi di dato di base, come i numeri interi o le stringhe, vi è un immediata corrispondenza, ad esempio

Stringa $\longleftrightarrow$ `varchar`

Tutti i tipi di dato specializzati, necessiteranno di un tipo di dato nuovo, utilizzando il costrutto `CREATE DOMAIN` di `SQL` , per i tipi di dato enumerativi, sarà necessario il costrutto `CREATE TYPE` .


```
CREATE TYPE Genere AS ENUM ('M','F');

CREATE DOMAIN Age AS integer
        CHECK (value >= 18 and value<=99)
```

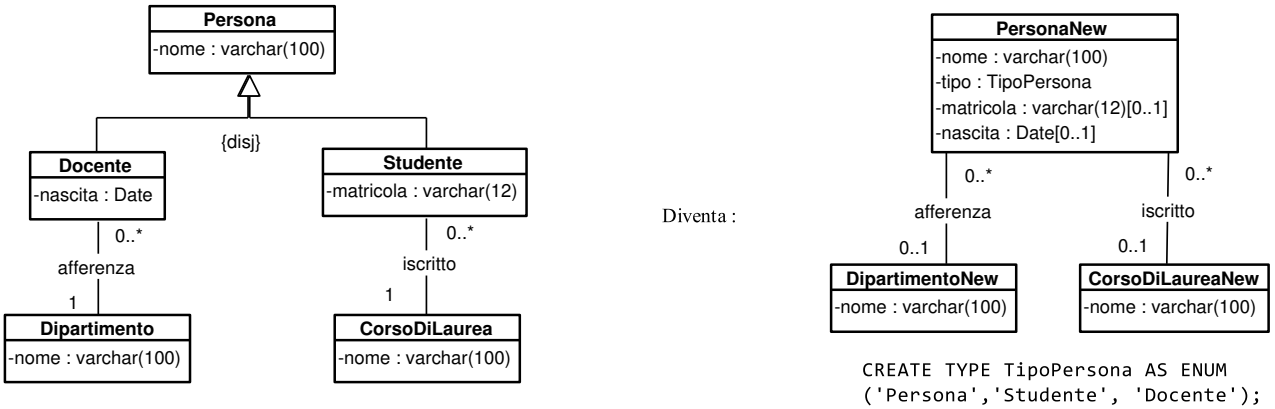
PASSO 3

Questo passo risulta il più "ostico", vogliamo modificare il nostro schema in modo che non contenga alcuna relazione *is-a*, ne fra classi, ne fra associazioni, dato che quest'ultime non sono supportate dal DBMS.

Iniziamo con le generalizzazioni fra classi, esistono tre diversi approcci per liberarsene, ogni approccio, favorisce la scrittura e la velocità di esecuzione di una certa categoria di query, sfavorendone altre, colui che esegue la ristrutturazione, deve saper riconoscere quale approccio può essere più efficace a seconda dell'operazione.

- **Fusione** : Tale approccio prevede, il collasso di un insieme di sottoclassi nella superclasse, aumentando l'insieme di attributi della superclasse, in modo tale da poter distinguere ogni istanza di tale classe (ormai unica).

  Sarà quindi l'insieme di attributi (o relazioni) di ogni istanza a definire se un oggetto appartiene ad una certa categoria o ad un altra (precedentemente, definito dalla relazione *is-a*). Dopo aver fuso le classi, è necessaria la definizione di nuovi necessari vincoli esterni.



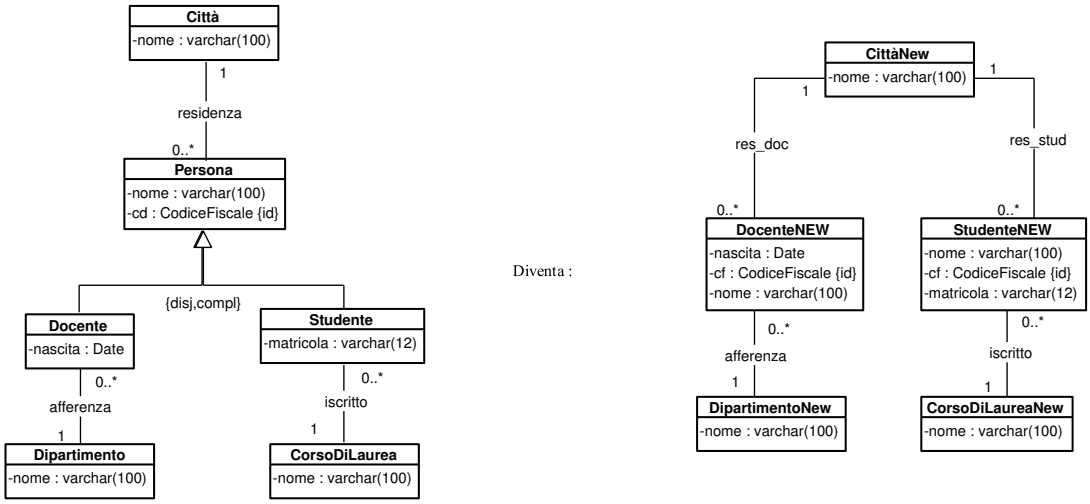

```
CREATE TYPE TipoPersona AS ENUM
('Persona','Studente', 'Docente');
```

Vanno considerati i nuovi vincoli esterni :

1. $\forall p \;\; PersonaNew(p) \rightarrow [tipo(p,'Studente') \leftrightarrow \exists m \; matricola(p,m)]$
2. $\forall p \;\; PersonaNew(p) \rightarrow [tipo(p,'Studente') \leftrightarrow \exists c \; iscritto(c,p)]$
3. $\forall p \;\; PersonaNew(p) \rightarrow [tipo(p,'Docente') \leftrightarrow \exists n \; nascita(p,n)]$
4. $\forall p \;\; PersonaNew(p) \rightarrow [tipo(p,'Docente') \leftrightarrow \exists d \; afferenza(d,p)]$

Quindi, l'approccio tramite fusione, comporta l'aggiunta di svariati vincoli esterni.

- **Divisione in classi disgiunte** : Tale approccio, prevede l'eliminazione di una superclasse, facendo vivere in maniera indipendente, quelle che prima erano sottoclassi, che dovranno ereditare tutti gli attributi e le associazioni, prima specificate nella superclasse, comportando ridondanza.



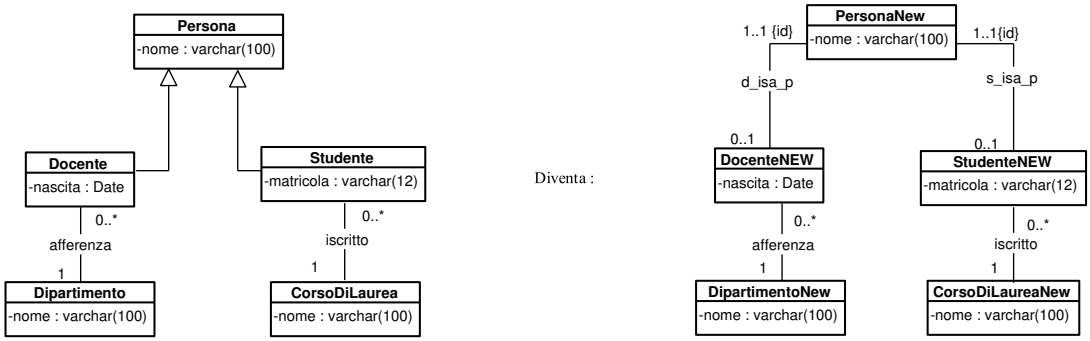
Riguardo l'esempio sovrastante : Se nel diagramma originale non potevano coesistere uno studente ed un docente con lo stesso codice fiscale, nel nuovo diagramma, ciò può accadere, è quindi necessario, quando vi sono degli identificatori nella superclasse eliminata, *definire dei nuovi vincoli* che impongano l'unicità delle istanze riguardo gli attributi univoci.

$$\neg\exists\ s,\ d,\ v\ \ StudenteNew(s)\ \wedge\ DocenteNew(d)\ \wedge\ cf(s,v)\ \wedge\ cf(d,v)$$

Questo approccio necessita quindi di vincoli per recuperare la semantica del vincolo di identificazione concettuale nelle superclassi eliminate.

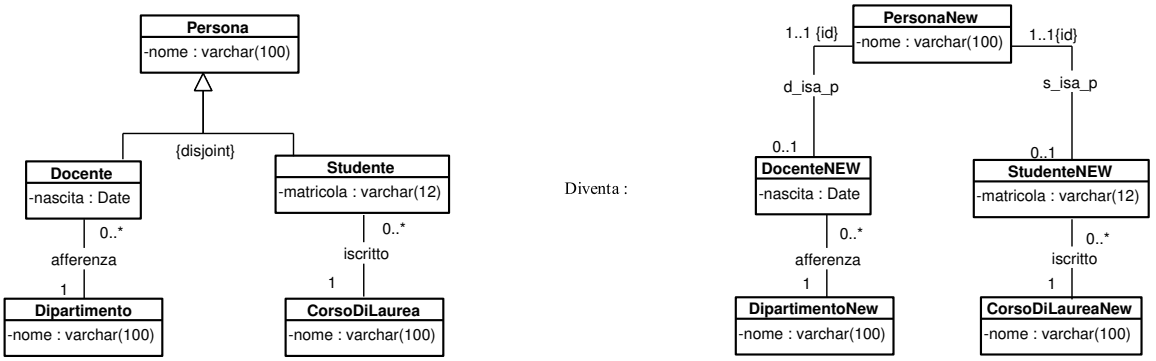
Tale approccio va utilizzato con cautela, se saranno necessarie query che comprendono la consultazione di tutte le istanze della superclasse ormai eliminata, sarà necessario eseguire un `JOIN` fra le istanze delle classi (ex-sottoclassi). Com'è di facile intuizione, tale approccio risulta sensato se applicato a generalizzazioni {disjoint,complete}.

- **Sostituzione con associazioni** : Quest'ultimo approccio risulta, nella pratica, il più utilizzato, in quanto riesce a mantenere la struttura del diagramma, il più simile possibile a quella del diagramma originale, si tratta del *sostituire* le relazioni *is-a* con delle associazioni.



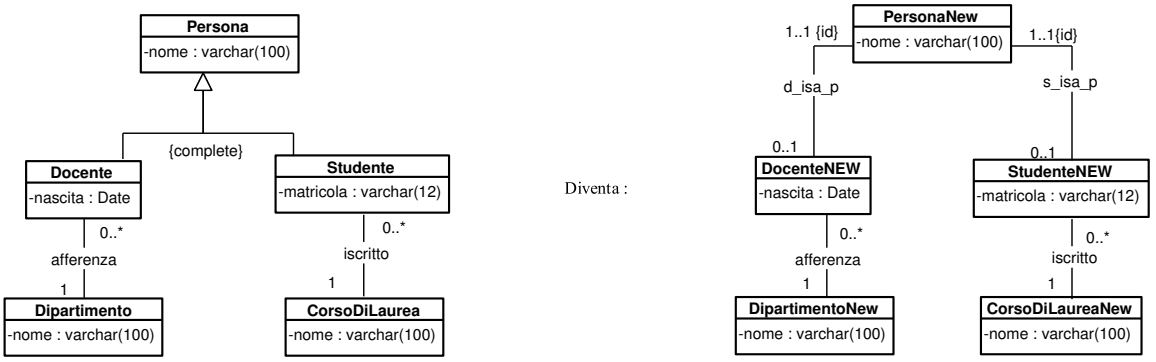
Si noti come la struttura sia perfettamente mantenuta, e sia necessario che le associazioni che modellano l'ex relazione *is-a*, comprendano il vincolo di identificazione (un docente non può essere due persone diverse).

Se la generalizzazione comprende dei vincoli complete e/o disjoint, è necessaria l'aggiunta di nuovi vincoli esterni, ad esempio, se la relazione dell'immagine sovrastante fosse disjoint, sarebbe necessario imporre la non-esistenza di un'istanza di PersonaNew, che ha un link sia verso un istanza di DocenteNew, sia verso un istanza di StudenteNew.



$$\neg\exists p,\ s,\ d$$
$$PersonaNew(p)\ \wedge\ StudenteNew(s)\ \wedge\ DocenteNew(d)$$
$$\wedge\ s\_isa\_p(s,p)\ \wedge\ d\_isa\_p(d,p)$$

Nel caso del vincolo complete, è necessario specificare che ogni istanza di PersonaNew, è almeno associata, o ad un'istanza di StudenteNew, oppure ad un istanza di DocenteNew.



$$\forall p\ PersonaNew(p) \rightarrow [\exists s\ s\_isa\_p(s,p)]\ \vee\ [\exists d\ d\_isa\_p(d,p)]$$

Com'è di facile intuizione, in una generalizzazione {disjoint,complete} saranno necessari entrambi i vincoli.