ITC INFORMATION AND COMMUNICATIONS TECHNOLOGY ACADEMY

MODULO: Progettazione
UNITÀ: Progettazione.1

Prof. Toni Mancini
Dipartimento di Informatica
Sapienza Università di Roma

Slide P.1

Design di applicazioni in Python

Passi della progettazione di un'applicazione in Python

A partire dallo schema concettuale dell'applicazione (output della fase di analisi), ovvero:

- diagramma UML concettuale delle classi
- specifiche dei tipi di dato
- specifiche delle classi
- specifiche dei vincoli esterni
- diagramma UML concettuale degli use-case
- specifiche concettuali degli use-case

effettueremo le seguenti attività:

1. Decideremo la corrispondenza dei tipi di dato concettuali in tipi di dato supportati da Python
2. Progetteremo l'architettura dell'applicazione Python tenendo conto di tutti i vincoli isolati in fase di analisi concettuale e dei requisiti di performance
3. Progetteremo le specifiche realizzative delle operazioni di use-case e delle operazioni di classe.

Ristrutturazione del diagramma UML delle classi

- **Obiettivo**: trasformare la *porzione* del diagramma UML concettuale delle classi relativa alle classi/associazioni le cui istanze si vogliono rappresentare come oggetti del programma in un diagramma equivalente, ma dalla cui struttura si possa direttamente derivare il codice.

- **La metodologia prevede una sequenza di passi da eseguire nell'ordine dato.** Ogni passo può prevedere alcune alternative da valutare caso per caso (con un occhio alle performance):

  1. Sostituzione dei tipi di dato concettuali con opportuni tipi supportati dal linguaggio di programmazione (tipi base, di libreria, o definiti dall'utente)
  2. Ristrutturazione delle generalizzazioni tra classi
  3. Ristrutturazione delle generalizzazioni tra associazioni

Il diagramma delle classi ristrutturato è, dal punto di vista concettuale, di pessima qualità.

Ma questo non è importante: la ristrutturazione è solo un modo grafico di procedere per ottenere un semi-lavorato equivalente all'originale e da cui facilmente produrre il codice del programma!

ITC INFORMATION AND COMMUNICATIONS TECHNOLOGY ACADEMY

MODULO: Progettazione
UNITÀ: Progettazione.1

Prof. Toni Mancini
Dipartimento di Informatica
Sapienza Università di Roma

Slide P.1

Design di applicazioni in Python

Ristrutturazione del diagramma UML delle classi

ITC INFORMATION AND COMMUNICATIONS TECHNOLOGY ACADEMY

MODULO: Progettazione
UNITÀ: Progettazione.1

Prof. Toni Mancini
Dipartimento di Informatica
Sapienza Università di Roma

Slide P.1

Progettazione del software
Design di applicazioni in Python
Ristrutturazione del diagramma UML delle classi

Progettazione e realizzazione dei tipi di dato

Progettazione dei tipi di dato

Obiettivo:

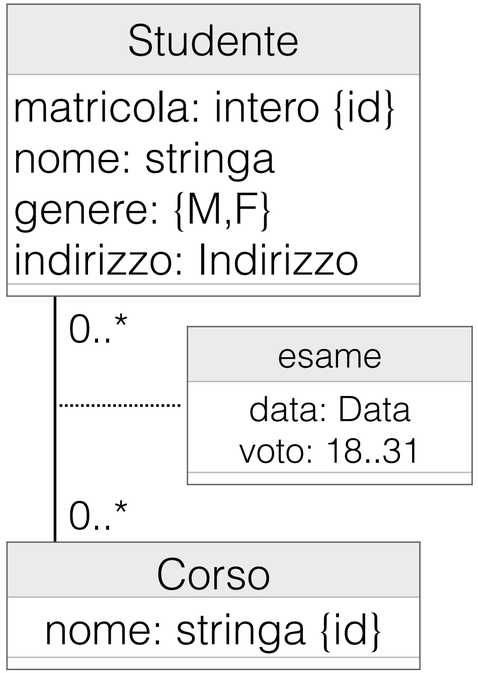
- Ristrutturare il diagramma UML concettuale delle classi in uno equivalente che contenga **solo attributi di tipi di dato supportati dal linguaggio di programmazione scelto**.

Metodologia:

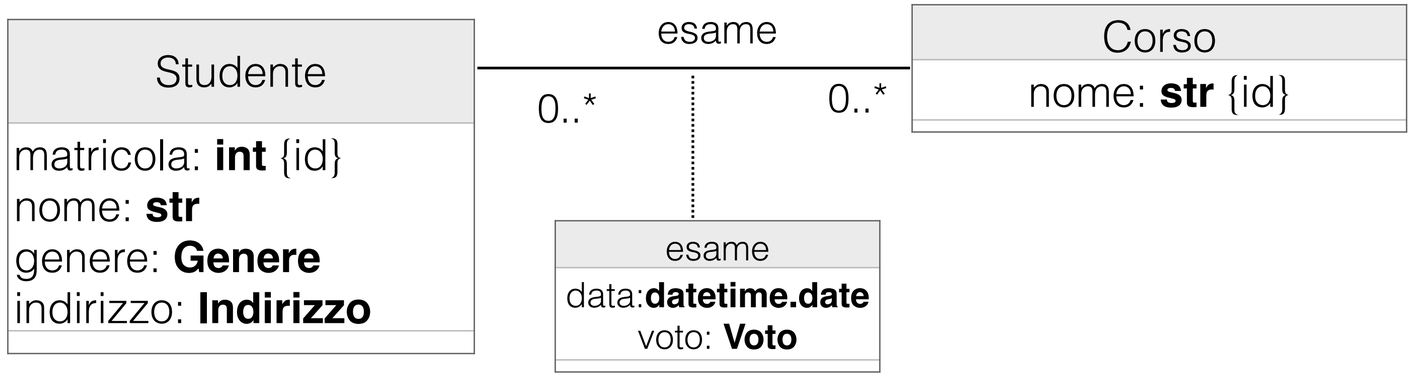
- Scegliere un opportuno tipo di dato supportato dal linguaggio per ogni attributo, anche definendo nuove class per rappresentarne le istanze (tipi di dato definiti dall'utente).
  - **Tipi concettuali base**: intero, reale, stringa, Booleano, DataOra, Data, Ora, etc.
    —> hanno un immediato corrispettivo in Python: int, float, str, bool, datetime, datetime.date, datetime.time, etc. [link a documentazione Python]
  - **Tipi concettuali enumerativi:** ad es., {M,F}, etc.
    —> utilizzeremo i tipi enumerativi disponibili in Python
  - **Tipi concettuali specializzati**: ad es., intero > 0, reale <= 0, x..y, etc. e **tipi concettuali definiti dall'utente o composti:** ad es., Indirizzo, etc.
    —> se possibile, utilizzare implementazioni del tipo disponibili nelle librerie esistenti
    —> altrimenti, definire ulteriori class Python (che non hanno alcun corrispettivo nel diagramma delle classi) le cui istanze rappresentano valori del tipo. Gli oggetti di queste class saranno immutabili e Python dovrà poter riconoscere se oggetti diversi rappresentano in realtà lo stesso valore

Esempio (tipi base, specializzati, enumerativi, composti):

Diagramma delle classi concettuale



Esito del passo di ristrutturazione



Tipo di dato Indirizzo: record(

- via: stringa
- civico: intero *# si può fare di meglio!*

)

Il tipo Genere sarà progettato come tipo enumerativo Python

I tipi Indirizzo e Voto saranno realizzati tramite nuove class Python (Indirizzo e Voto) i cui oggetti (immutabili) rappresenteranno i valori dei rispettivi tipi (Python dovrà poter riconoscere se oggetti diversi rappresentano lo stesso valore)

Realizzazione dei tipi di dato: tipi di dato enumerativi

Studente
matricola: int {id} nome: str genere: **Genere** indirizzo: Indirizzo

```
from enum import *

class Genere(StrEnum):
    uomo = auto()
    donna = auto()
```

esame
data:datetime.date voto: **Voto**

Class Python di oggetti immutabili

Eredita da tipo base int

```
class Voto(int):
    def __new__(cls, v:int|float|Self)->Self:
        if v < 18 or v > 30:
            raise ValueError(f"Value v == {v} must be between 18 and 30")
        return int.__new__(cls, v)
```

Realizzazione dei tipi di dato: tipi di dato composti

Studente
matricola: int {id} nome: str genere: Genere indirizzo: **Indirizzo**

Class Python di oggetti immutabili

Non possiamo ereditare da tipo base Python, quindi dobbiamo fare in modo che Python riconosca che due oggetti rappresentano in realtà lo stesso valore: **__hash__()** ed **__eq__()**

```
class Indirizzo:
  # campi dati:
  _via:str
  _civico:…
  def __init__(self, via:str, civ:…):
    if … i valori non sono legali …:
      raise ValueError(…)
    self._via = via
    self._civico = civ
    …
  def via(self)->str:
    return self._via
  def civico(self)->…: …
```

```
# class Indirizzo implementa un tipo di dato: Python deve riconoscere
se oggetti diversi rappresentano lo stesso valore
def __hash__(self)->int:
  return hash( (self.via(), self.civico()) )
def __eq__(self, other:Any)->bool:
  if other is None or \
          not isinstance(other, type(self)) or \
          hash(self) != hash(other):
    return False
  return (self.via(), self.civico() ) == (other.via(), other.civico())
```

Scelta dei tipi di dato (cont.)

Al termine del passo di definizione della corrispondenza dei tipi di dato concettuali in tipi supportati dal linguaggio in uso, tutti gli attributi del diagramma ristrutturato hanno un dominio supportato dal linguaggio

ITC INFORMATION AND COMMUNICATIONS TECHNOLOGY ACADEMY

MODULO: Progettazione
UNITÀ: Progettazione.1

Prof. Toni Mancini
Dipartimento di Informatica
Sapienza Università di Roma

Slide P.1

Progettazione del software
Design di applicazioni in Python

Ristrutturazione del diagramma UML delle classi

Generalizzazioni

Ristrutturazione delle generalizzazioni

Obiettivo:

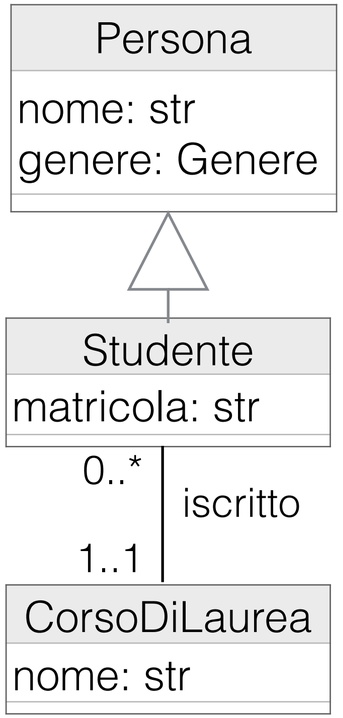
- Ristrutturare il diagramma UML concettuale delle classi in uno equivalente che **non contenga:**
  - **Generalizzazioni <u>non disgiunte</u> tra classi o associazioni**
    - Difatti, un oggetto Python deve appartenere ad una ed una sola classe più specifica
  - **Generalizzazioni tra classi o associazioni le cui istanze (oggetti o link) possono cambiare la loro classe/associazione più specifica durante la loro vita**
    - Difatti, un oggetto Python non può cambiare la sua classe più specifica una volta creato

Metodologia:

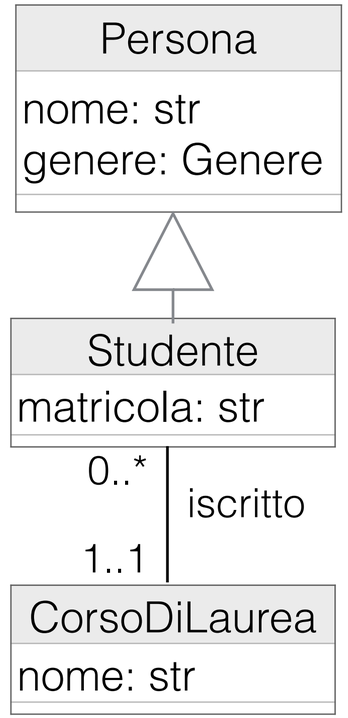
- **Relazioni is-a o generalizzazioni tra classi problematiche**:
  - **Fusione** delle sottoclassi nella loro superclasse
- **Relazioni is-a o generalizzazioni tra associazioni:** due possibili approcci:
  - **Fusione** di sotto-associazioni nelle loro super-associazioni, se opportuno a valle del passo precedente
  - Aggiunta di **vincoli esterni** (che saranno opportunamente gestiti più avanti)

Gli oggetti **non** hanno bisogno di cambiare la loro classe più specifica

Diagramma delle classi di partenza



Esito del passo di ristrutturazione

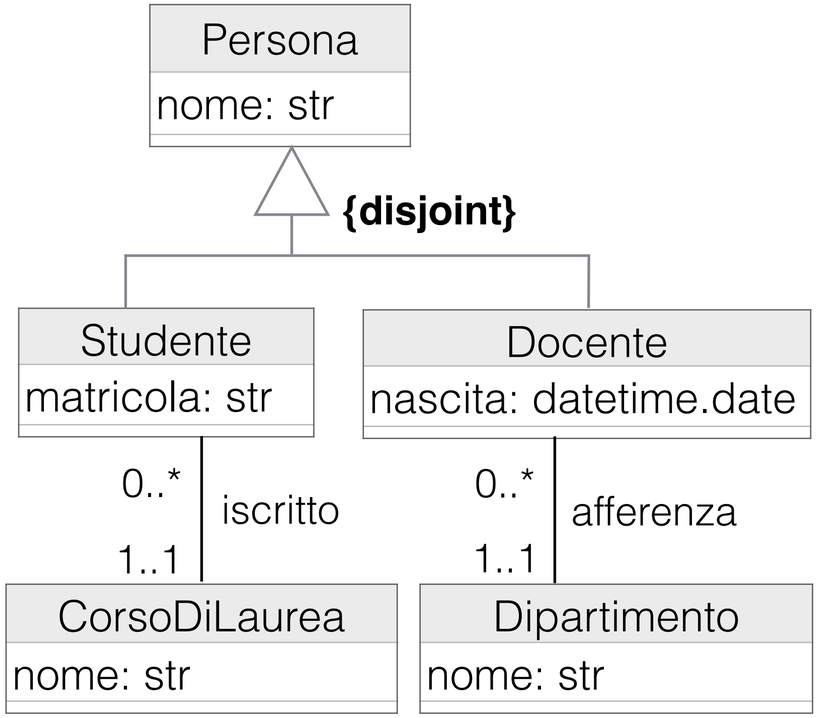


Nota: abbiamo appurato che, una volta creato, un oggetto di classe più specifica Persona non ha bisogno di diventare di classe Studente e viceversa

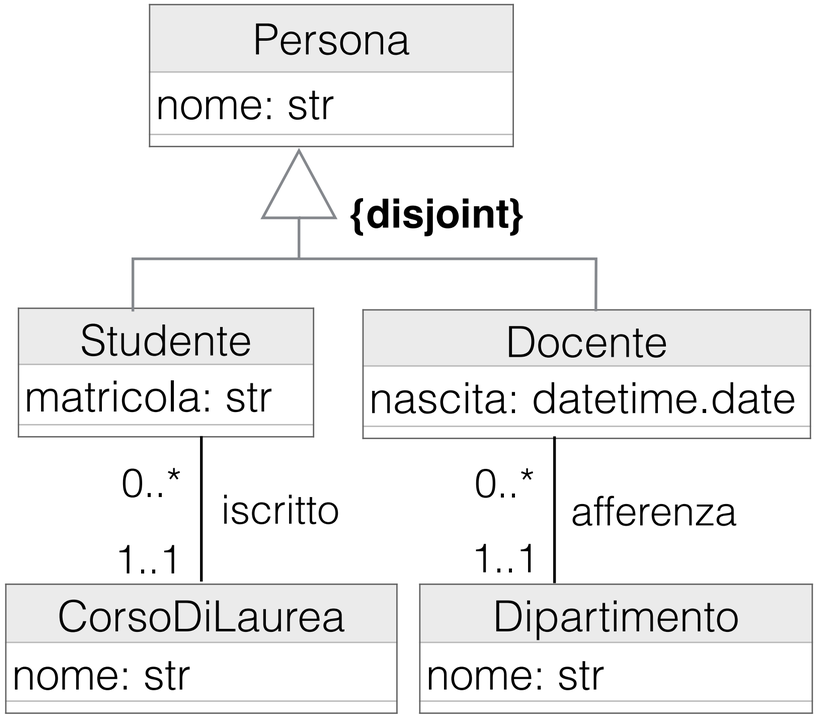
Nessuna azione necessaria

Le classi sono **disgiunte** e gli oggetti **non** hanno bisogno di cambiare la loro classe più specifica

Diagramma delle classi di partenza



Esito del passo di ristrutturazione



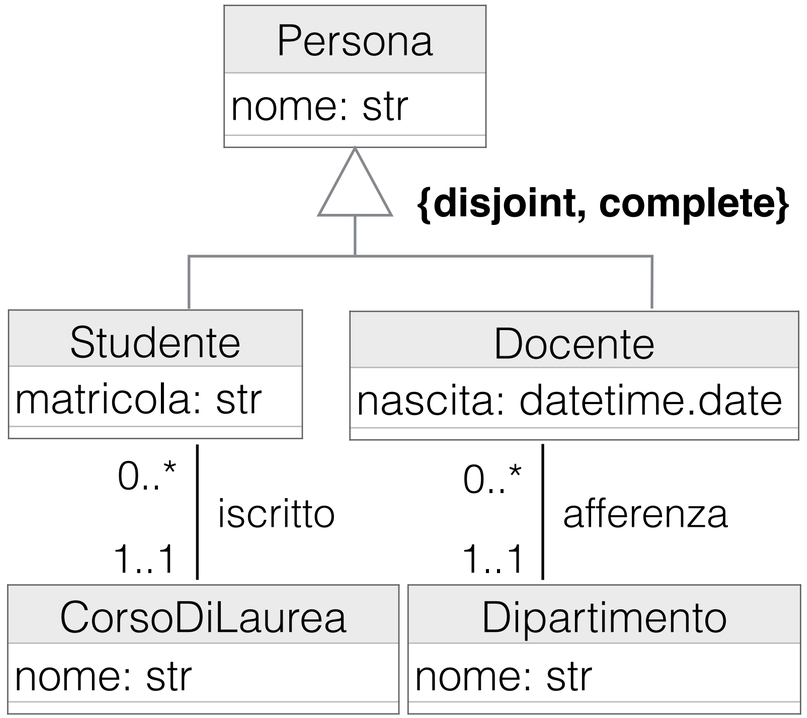
Nota: abbiamo appurato che, una volta creato:

- un oggetto di classe più specifica Persona non ha bisogno di diventare di classe Studente né Docente
- un oggetto di classe Studente o Docente non ha bisogno di diventare un oggetto di classe più specifica Persona
- un oggetto di classe Studente (o Docente) non ha bisogno di trasformarsi in un oggetto di classe Docente (o Studente)

Nessuna azione necessaria

Le classi sono **disgiunte** e gli oggetti **non** hanno bisogno di cambiare la loro classe più specifica

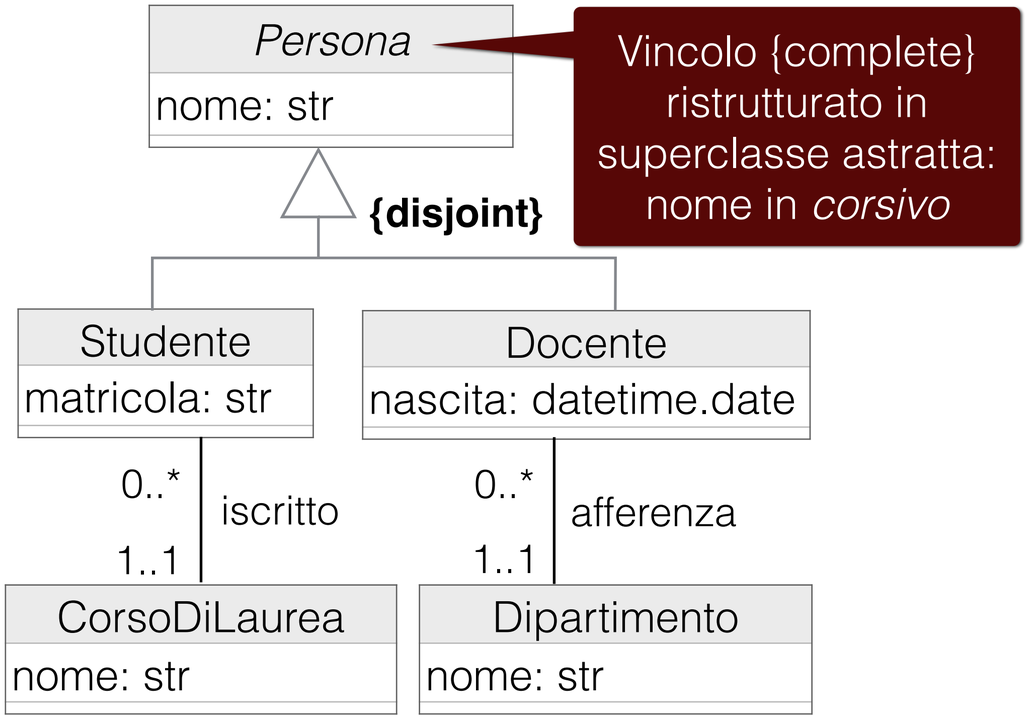
Diagramma delle classi di partenza



Nota: abbiamo appurato che, una volta creato:

- un oggetto di classe più specifica Persona non ha bisogno di diventare di classe Studente né Docente
- un oggetto di classe Studente non ha bisogno di trasformarsi in un oggetto di classe Docente, e viceversa

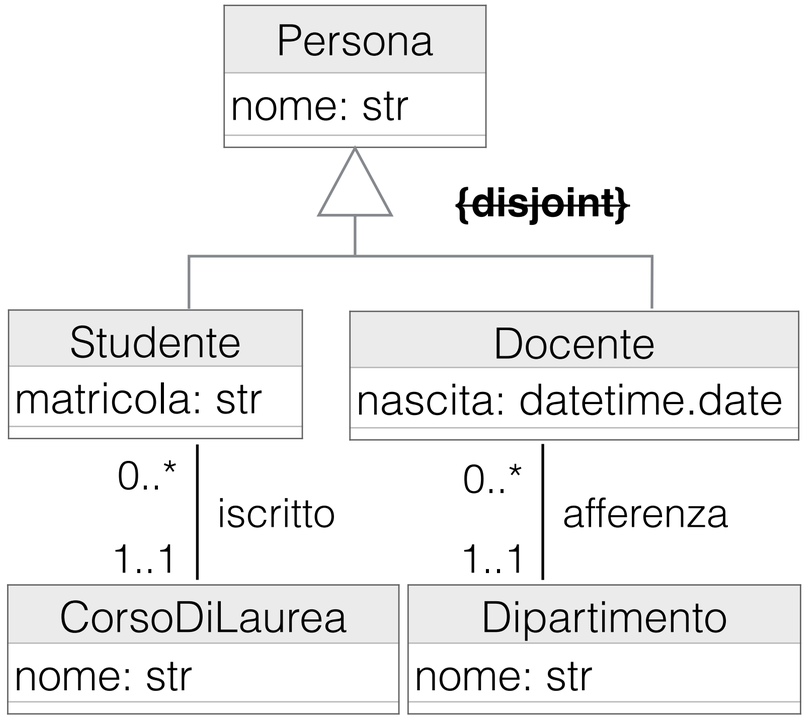
Esito del passo di ristrutturazione



Generalizzazione **complete**:
la classe Persona viene ristrutturata come **astratta**

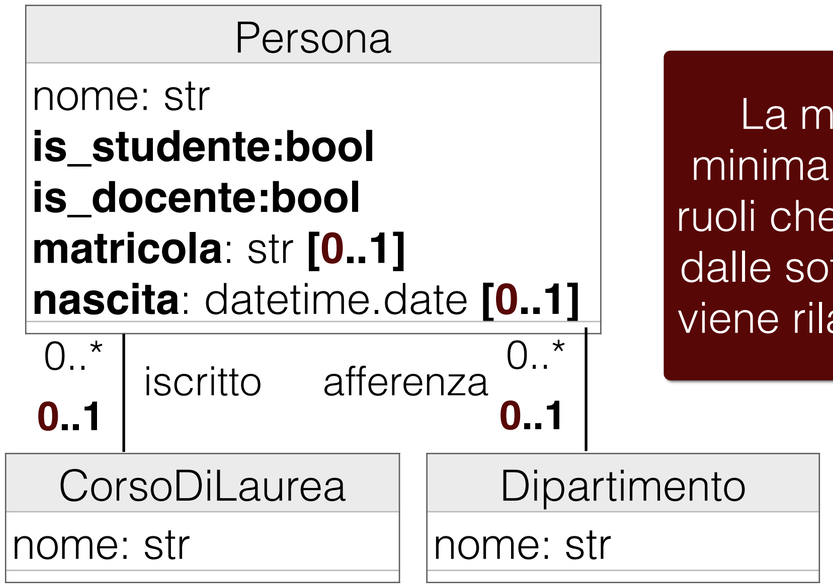
Le classi **non** sono **disgiunte oppure** gli oggetti **hanno bisogno** di **cambiare** la loro classe più

Diagramma delle classi di partenza



Fondiamo le sottoclassi nella superclasse

Esito del passo di ristrutturazione



La molteplicità minima di attributi e ruoli che provengono dalle sottoclassi fuse viene rilassata a zero

Nota: generalizzazione non disjoint oppure abbiamo appurato che, una volta creato:

- un oggetto di classe più specifica Persona può dover diventare di classe Studente o Docente
- un oggetto di classe Studente può trasformarsi in un oggetto di classe Docente, e viceversa

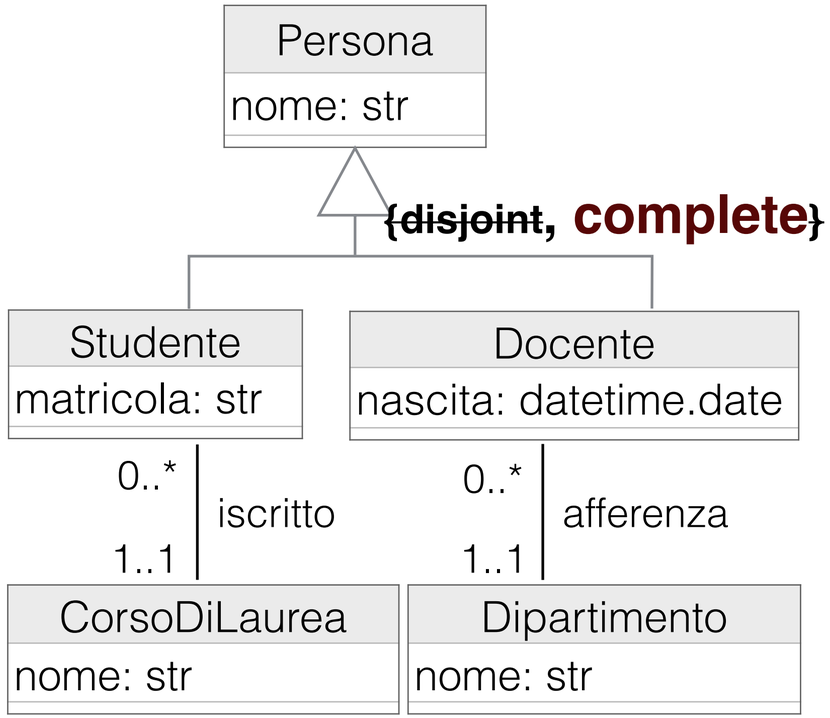
Vincoli esterni: per ogni istanza p di Persona:

1. p.is_studente = TRUE se e solo se p.matricola è valorizzato
2. p.is_studente = TRUE se e solo se p è coinvolto in un link “iscritto”
3. p.is_docente = TRUE se e solo se p.nascita è valorizzato
4. p.is_docente = TRUE se e solo se p è coinvolto in un link “afferenza”

Implementano i requisiti persi nella fusione

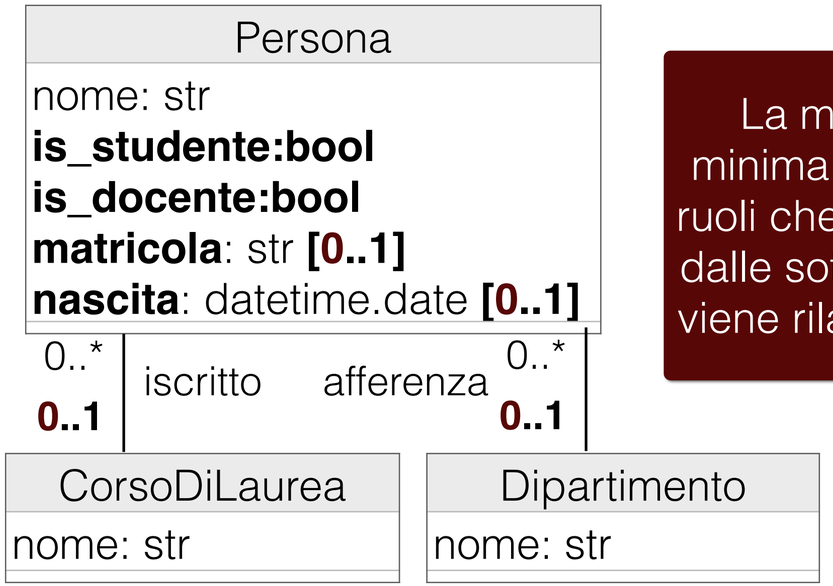
Le classi **non** sono **disgiunte oppure** gli oggetti **hanno bisogno** di **cambiare** la loro classe più specifica

Diagramma delle classi di partenza



Fondiamo le sottoclassi nella superclasse

Esito del passo di ristrutturazione



La molteplicità minima di attributi e ruoli che provengono dalle sottoclassi fuse viene rilassata a zero

Nota: generalizzazione non disjoint oppure abbiamo appurato che, una volta creato:

- un oggetto di classe più specifica Persona può dover diventare di classe Studente o Docente
- un oggetto di classe Studente può trasformarsi in un oggetto di classe Docente, e viceversa

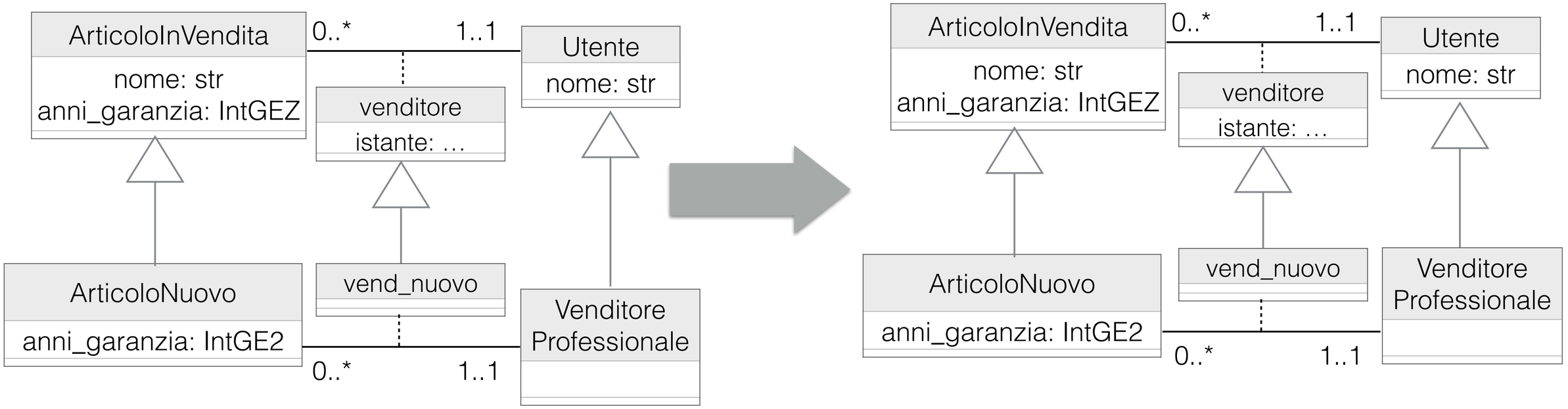
Vincoli esterni: per ogni istanza p di Persona:

1. p.is_studente = TRUE se e solo se p.matricola è valorizzato
2. p.is_studente = TRUE se e solo se p è coinvolto in un link “iscritto”
3. p.is_docente = TRUE se e solo se p.nascita è valorizzato
4. p.is_docente = TRUE se e solo se p è coinvolto in un link “afferenza”
5. p.is_docente = TRUE **oppure** p.is_studente = TRUE

Implementa {complete}

Ristrutturazione delle generalizz. tra associazioni

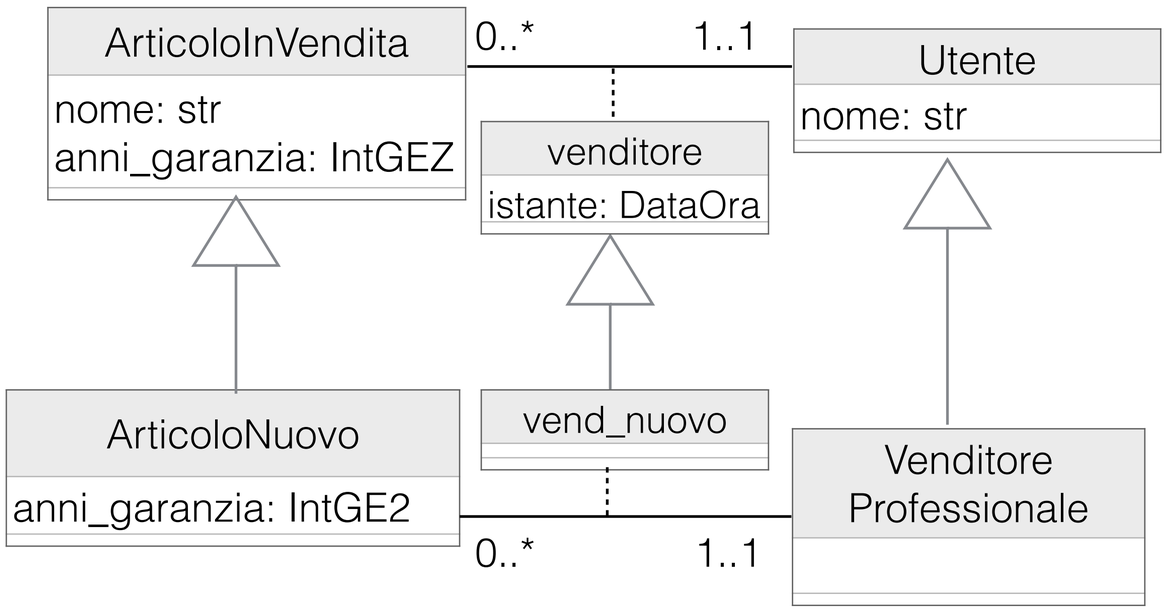
Ogni articolo è venuto da uno o più utenti. Gli articoli nuovi sono venduti da almeno un venditore professionale.

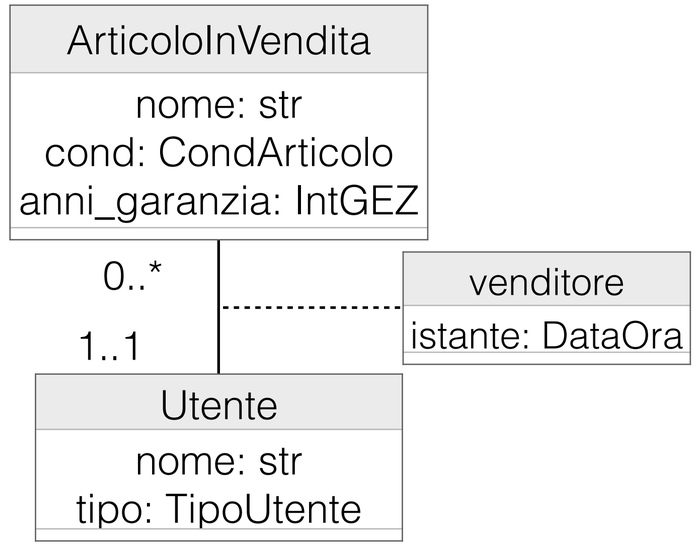


Nessuna azione necessaria

Ristrutturazione delle generalizz. tra associazioni: fusione

Ristrutturazione **in caso le generalizzazioni tra classi siano state ristrutturate per fusione**





I tipi CondArticolo e TipoUtente saranno progettati come tipi enumerativi Python

Vincoli esterni:

per ogni a:ArticoloInVendita:

- se a.cond = ‘nuovo’ allora a.anni_garanzia è di tipo IntGE2
- se a.cond = ‘nuovo’ allora il link (a,u):venditore nel quale ‘a’ è coinvolto è tale che u.tipo = ‘prof’

Ristrutturazione delle generalizzazioni (cont.)

Al termine del passo di ristrutturazione delle generalizzazioni, tutte le classi e le associazioni del diagramma ristrutturato sono direttamente implementabili nel linguaggio di programmazione scelto.

ITC INFORMATION AND COMMUNICATIONS TECHNOLOGY ACADEMY

MODULO: Progettazione
UNITÀ: Progettazione.1

Prof. Toni Mancini
Dipartimento di Informatica
Sapienza Università di Roma

Slide P.1

Progettazione del software
Design di applicazioni in Python

Ristrutturazione del diagramma UML delle classi

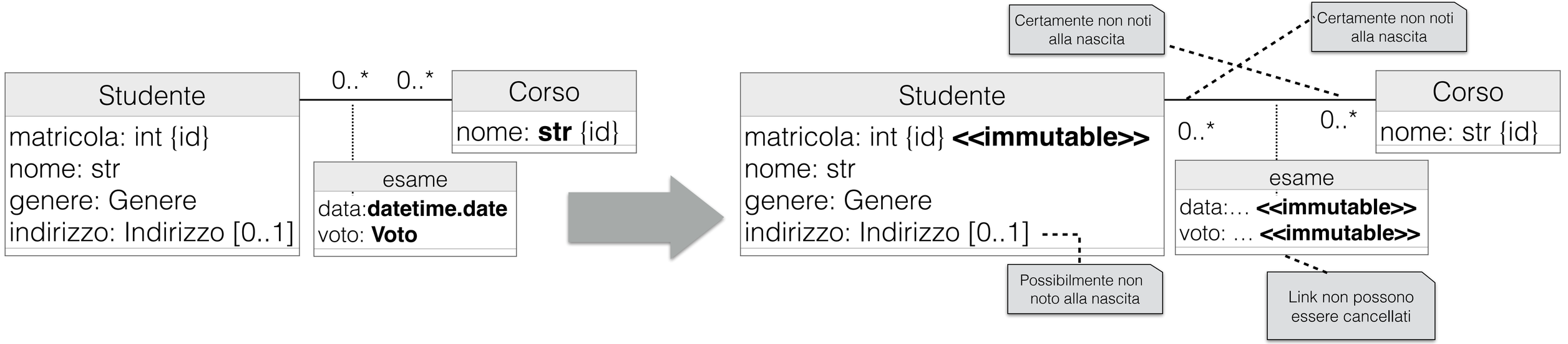
Evoluzione delle proprietà

Evoluzione delle proprietà

- In generale, le proprietà (attributi e link) di un oggetto UML evolvono in maniera arbitraria durante il suo ciclo di vita
- Esistono però alcuni casi particolari che vanno presi in considerazione nella fase di design.
- Definiamo una proprietà
  - **Mutabile**
    se il suo valore può mutare arbitrariamente durante il ciclo di vita dell'oggetto (fermo restando il soddisfacimento di eventuali vincoli sui valori ammessi)
  - **Immutabile**
    se, una volta che è stata specificata, il suo valore resta invariato per tutto il ciclo di vita dell'oggetto
  - **Mutabile ad evoluzione vincolata**
    se non può mutare arbitrariamente, ma solo soddisfacendo delle condizioni (ad es., il valore può solo crescere)
  - **Certamente nota alla nascita**
    se deve essere conosciuta nel momento in cui nasce l'oggetto
  - **Possibilmente non nota alla nascita**
    se può essere sconosciuta nel momento in cui nasce l'oggetto (deve avere vincolo di molt. [0..1])
  - **Certamente non nota alla nascita**
    se è certamente sconosciuta nel momento in cui nasce l'oggetto (deve avere vincolo di molt. [0..1])

- Tali caratteristiche degli attributi e delle associazioni sono da evincersi dallo schema concettuale (in particolare dalle operazioni di classe e use-case), oppure sono conseguenti ad altre scelte effettuate in fase di analisi o di design

Evoluzione delle proprietà: esempio



Classe Studente:

- attributo *matricola*: **immutabile**, valore certamente noto alla nascita dell'oggetto (a causa del vincolo di molt. [1..1])
- attributo *nome*: mutabile, noto alla nascita ([1..1])
- attributo *genere:* mutabile, noto alla nascita ([1..1])
- attributo *indirizzo*: mutabile, **poss. non** noto alla nascita
- link di associazione *esame* che coinvolgono ogni oggetto di classe Studente: mutabili, **certamente non** noti alla nascita

Classe Corso:

- *nome*: mutabile, noto alla nascita ([1..1])
- link di assoc. *esame* che coinvolgono ogni oggetto di classe Corso: mutabili, **certamente non** noti alla nascita

Associazione esame:

- data: **immutabile**, noto alla nascita ([1..1])
- voto: **immutabile**, noto alla nascita ([1..1])
- link possono essere creati, ma non cancellati

Evoluzione delle proprietà: assunzioni di default

Per mantenere il diagramma delle classi ristrutturato leggibile e velocizzare la fase di design, adotteremo le seguenti assunzioni di default.

Distinguiamo innanzitutto fra:

- **proprietà singole**
  ovvero attributi (di classe o di associazione) e associazioni (o meglio, ruoli) con molteplicità **1..1**
- **proprietà con molteplicità minima pari a zero**
  ovvero attributi di classe o di associazione) e associazioni (o meglio, ruoli) con molteplicità **0..1, 0..***, **0..2**, etc.

Le nostre **assunzioni di default**, ovvero che valgono in assenza di ulteriori elementi, sono:

- **tutte** le proprietà sono **mutabili**
- le **proprietà singole** (1..1) o quelle con **molteplicità minima pari a 1** (ad 1..*) sono **note alla nascita** (ovviamente, data la molteplicità 1..1)
- le **proprietà con molt. minima 0** sono **possibilmente non note alla nascita**.

Contrassegnamo nel diagramma (con <<immutable>> o nota UML) solo i valori diversi dai default.

ITC INFORMATION AND COMMUNICATIONS TECHNOLOGY ACADEMY

MODULO: Progettazione
UNITÀ: Progettazione.1

Prof. Toni Mancini
Dipartimento di Informatica
Sapienza Università di Roma

Slide P.1

Progettazione del software
Design di applicazioni in Python

Ristrutturazione del diagramma UML delle classi

Visibilità delle proprietà

Visibilità delle proprietà

- Al fine di aumentare l'information hiding, vanno decise quali proprietà (attributi e operazioni) delle classi e associazioni possono essere accessibili dall'esterno e quali no

- I livelli di visibilità possibili sono tre:
  - **pubblico (+)**
    La proprietà è accessibile anche dal di fuori della classe/associazione dove è definita
  - **protetto (#)**
    La proprietà è accessibile solo nella classe/associazione dove è definita e nelle sue sottoclassi/ sotto-associazioni
  - **privato (-)**
    La proprietà è accessibile solo nella classe/associazione dove è definita

Visibilità delle proprietà: criteri

I criteri che vanno usati per decidere il livello di visibilità delle proprietà sono i seguenti:

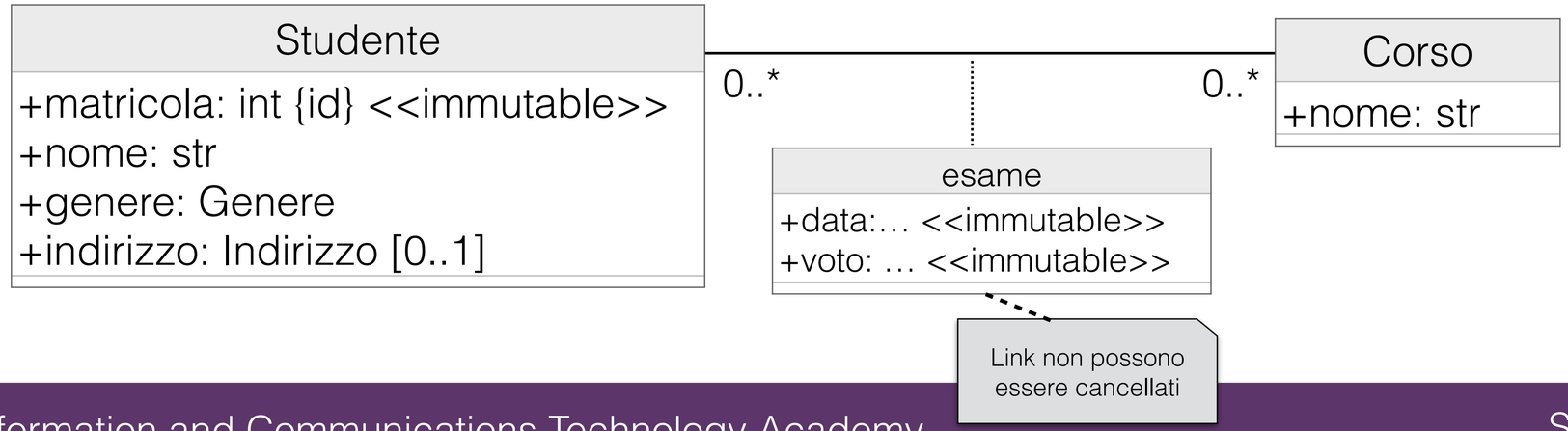
Attributi

- Gli attributi definiti in fase di Analisi sono pubblici (+)
- Gli attributi aggiunti in fase di Design, durante il passo di ristrutturazione delle generalizzazioni per fusione, che definiscono la natura dei singoli oggetti (ad es., is_docente:bool, is_studente:bool) sono pubblici (+)
- Gli altri attributi (di supporto) aggiunti in fase di Design sono privati (-) oppure protetti (#) se è necessario accedervi da almeno una sottoclasse

Operazioni di classe

- Le operazioni definite in fase di Analisi sono pubbliche
- Le operazioni di supporto definite in fase di Design (ad es., quelle aggiunte per facilitare la specifica realizzativa delle operazioni di classe) sono private (o protette)

In generale, per stabilire se una proprietà debba essere dichiarata protetta (#) invece che privata (-), vanno considerate le specifiche realizzative prodotte, da dove si possono evincere le necessità, da parte delle sottoclassi, di accedere tale proprietà.



ITC INFORMATION AND COMMUNICATIONS TECHNOLOGY ACADEMY

MODULO: Progettazione
UNITÀ: Progettazione.1

Prof. Toni Mancini
Dipartimento di Informatica
Sapienza Università di Roma

Slide P.1

Progettazione del software
Design di applicazioni in Python
Ristrutturazione del diagramma UML delle classi

Specifica delle operazioni di classe e di use-case

Ristrutturazione delle specifiche di classi e di use-case

Obiettivo:

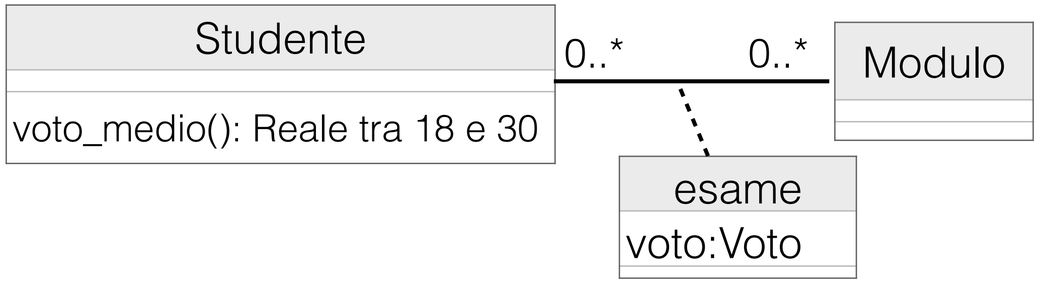
- Ristrutturare le specifiche concettuali delle classi e degli use-case in nuove specifiche sul diagramma UML delle classi ristrutturato.

Metodologia:

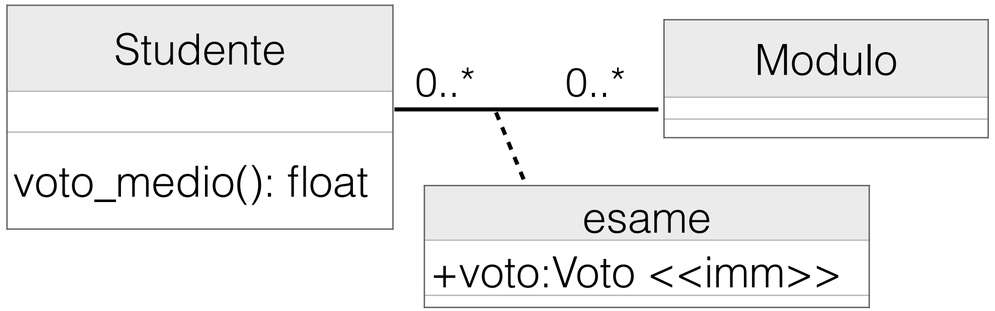
- Definire un algoritmo (in forma di pseudo-codice) per ogni operazione di classe o di use-case che verifichi le sue precondizioni e, se soddisfatte, garantisca il raggiungimento delle sue post-condizioni.

Ristrutturazione delle specifiche di classi e di use-case (cont.)

Diagramma delle classi concettuale



Esito del passo di ristrutturazione



Specifica concettuale della classe Studente

voto_medio(): Reale tra 18 e 30

pre-condizioni:

Lo studente this è coinvolto in almeno un link di associazione esame.

post-condizioni:

- Sia E l'insieme dei link di associazione esame che coinvolgono lo studente this.
- Sia S la somma dei valori dell'attributo voto dei link di E
- result = S / |E| (dove |E| è il numero di elementi di E)

Specifica realizzativa della classe Studente

voto_medio(): float **| None**

algoritmo:

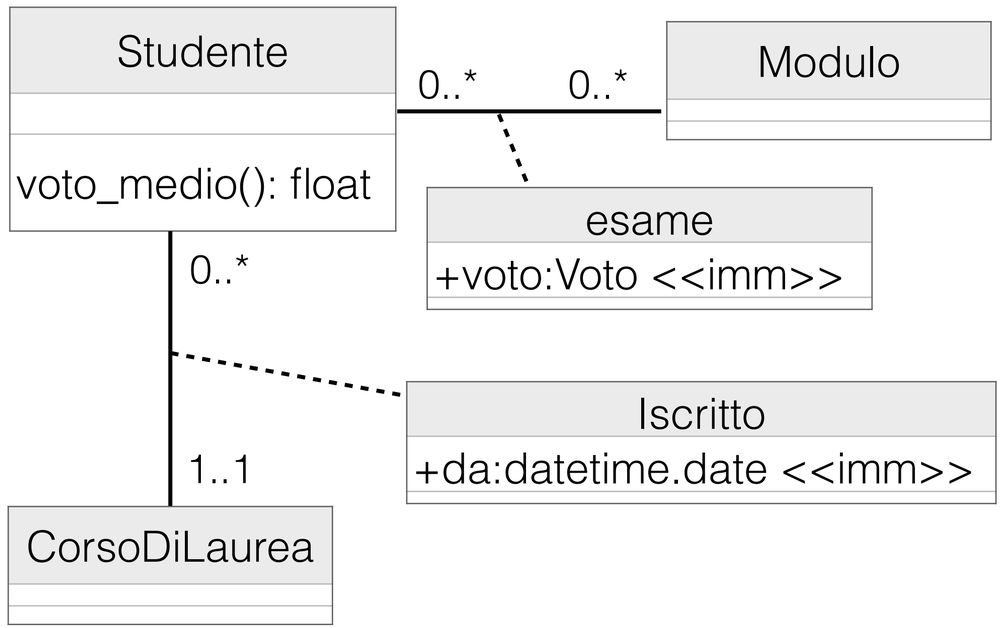
```
se |this.esame| == 0: return None
s = 0
altrimenti:
  per ogni l in this.esame:
     s = s + l.voto
  return s / |this.esame|
```

"this.esame": insieme di link di associazione esame che coinvolgono this

Per semplicità restituiamo un float, ma garantiamo per costruzione che sarà sempre tra 18 e 30

Decidiamo di segnalare la violazione delle precondizioni restituendo None piuttosto che generando un errore

Ristrutturazione delle specifiche di classi e di use-case (cont.)



Accesso ai valori degli attributi, alle operazioni e ai link che coinvolgono un oggetto

Assumendo che s denoti un oggetto di classe Studente, nello pseudo-codice possiamo scrivere:

- **s.voto_medio()** per denotare il valore dell'operazione invocata su s
- **s.esame** per denotare l'insieme dei link l dell'associazione esame che coinvolgono s.
  Per ognuno di tali link l = (s:Studente, m:Modulo):
  - **l.studente** denota lo studente di l ("studente" è il nome —di default in questo caso— del ruolo dell'associazione)
  - **l.modulo** denota il modulo di l
  - **l.voto** denota il valore dell'attributo voto di l
- **Scorciatoia**: se l'associazione ha vincolo di molt. 1..1, s.associazione denoterà il singolo link, non l'insieme formato da tale singolo link
- **Esempio: s.iscritto** denoterà il singolo link di associazione iscritto che coinvolge s (e non l'insieme formato da tale singolo link).
  Questo ci permette di denotare:
  - con **s.iscritto.corso_di_laurea** il singolo corso di laurea a cui è iscritto s
  - con **s.iscritto.da** il valore dell'attributo 'da' di tale singolo link

Ristrutturazione delle specifiche di classi e di use-case(cont.)

Al termine del passo di ristrutturazione della specifica delle operazioni di classe e di use case, le specifiche delle operazioni sono ora espresse in termini di algoritmi e sono coerenti con la nuova struttura del diagramma UML delle classi ristrutturato.

ITC INFORMATION AND COMMUNICATIONS TECHNOLOGY ACADEMY

MODULO: Progettazione
UNITÀ: Progettazione.1

Prof. Toni Mancini
Dipartimento di Informatica
Sapienza Università di Roma

Slide P.1

Progettazione del software
Design di applicazioni in Python

Ristrutturazione del diagramma UML delle classi

Specifica dei vincoli esterni

Vincoli esterni

Obiettivo:

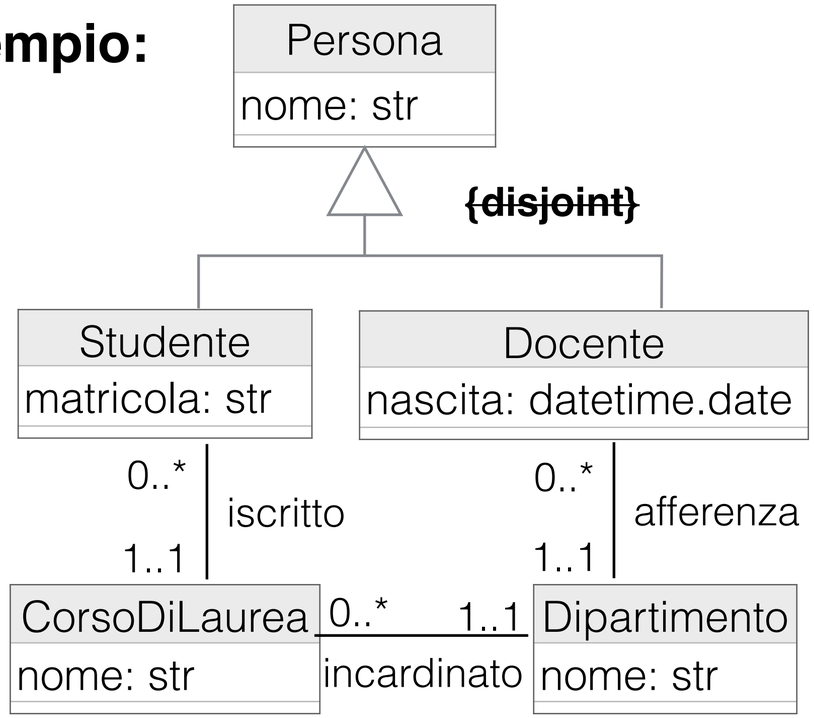
- Ristrutturare i vincoli esterni al diagramma UML concettuale delle classi in vincoli equivalenti sul diagramma UML delle classi ristrutturato.

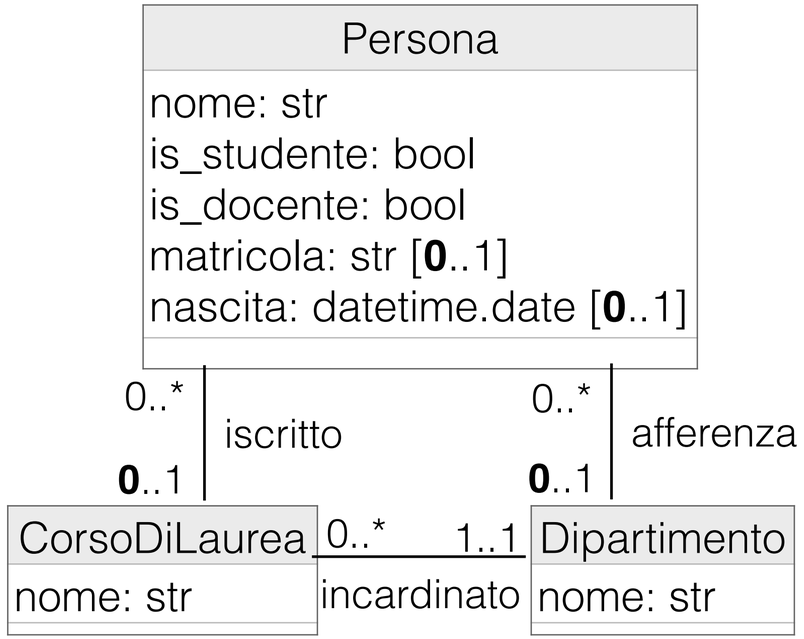
Metodologia:

- Aggiungere al documento di **specifica ristrutturata dei vincoli esterni** i vincoli esterni definiti, in fase di analisi, nelle specifiche concettuali delle classi e nella specifica concettuale dei vincoli esterni, opportunamente adattati alla struttura del diagramma UML delle classi ristrutturato.

Vincoli esterni (cont.)

Esempio:





Vincolo esterno [V.Docente.non_studia_in_proprio_cdl]:

Un docente non può essere anche studente in un corso di laurea incardinato dipartimento a cui afferisce. Formalmente:

Per ogni p:Docente tale che è vero anche p:Studente:

- sia c:CorsoDiLaurea tale che (p,c):iscritto
- sia d:Dipartimento tale che (p,d):afferenza

Non deve essere (c,d):incardinato.

Vincoli esterni:

[Da fusione] Per ogni p:Persona:

- p.is_studente = TRUE se e solo se matricola è valorizzato
- p.is_studente = TRUE se e solo se p è coinvolto in un link di assoc. iscritto
- p.is_docente = TRUE se e solo se nascita è valorizzato
- p.is_docente = TRUE se e solo se p è coinvolto in un link di assoc. afferenza

- **[V.Docente.non_studia_in_proprio_cdl]** Per ogni ogg. p:Persona tale che p.is_studente = TRUE e p.is_docente = TRUE, deve essere:

  p.iscritto.corso_di_laurea.incardinato.dipartimento != p.afferenza.dipartimento

ITC INFORMATION AND COMMUNICATIONS TECHNOLOGY ACADEMY

MODULO: Progettazione
UNITÀ: Progettazione.1

Prof. Toni Mancini
Dipartimento di Informatica
Sapienza Università di Roma

Slide P.1

Progettazione del software
Design di applicazioni in Python

Ristrutturazione del diagramma UML delle classi

Design delle associazioni: responsabilità

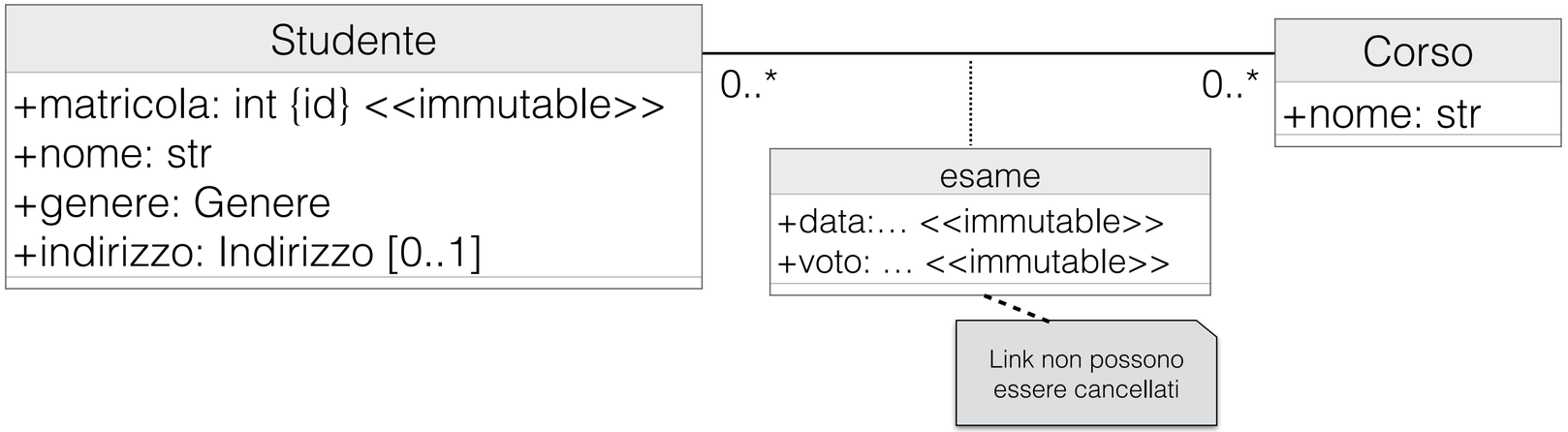
Design di associazioni: responsabilità

- Per ogni associazione, dobbiamo decidere quali delle due classi coinvolte ne hanno responsabilità
- Una classe C ha responsabilità su una associazione A (nella quale è coinvolta) se un oggetto c:C deve poter:
  - conoscere in quali link di A è coinvolto
  - aggiungere, eliminare, modificare i link nei quali è coinvolto
- Per ogni associazione, almeno una delle due classi coinvolte deve esserne responsabile

Una classe C ha responsabilità in una associazione A se:

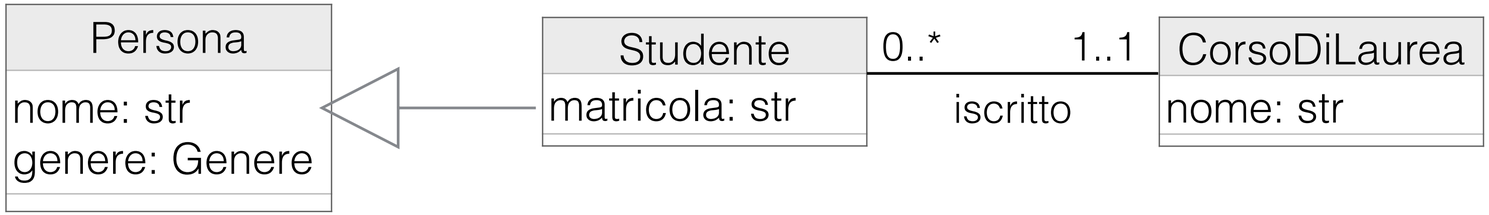
- Esiste una operazione di classe/use-case il cui algoritmo necessita che un oggetto c:C possa accedere/aggiungere/eliminare/modificare i suoi link di associazione A
- Partecipa ad A con vincoli di molteplicità diversi da 0..* (senza responsabilità su A, un oggetto c:C non potrebbe assicurarsi di soddisfare i vincoli di molteplicità)

Design di associazioni, responsabilità: esempio



- Una operazione di use-case chiede di calcolare la media dei voti del dato studente:
  - la classe Studente deve avere responsabilità sull'associazione esame
- Una operazione di use-case chiede di calcolare quanti studenti hanno superato un dato corso:
  - la classe Corso deve avere responsabilità sull'associazione esame

Design di associazioni, responsabilità: esempio

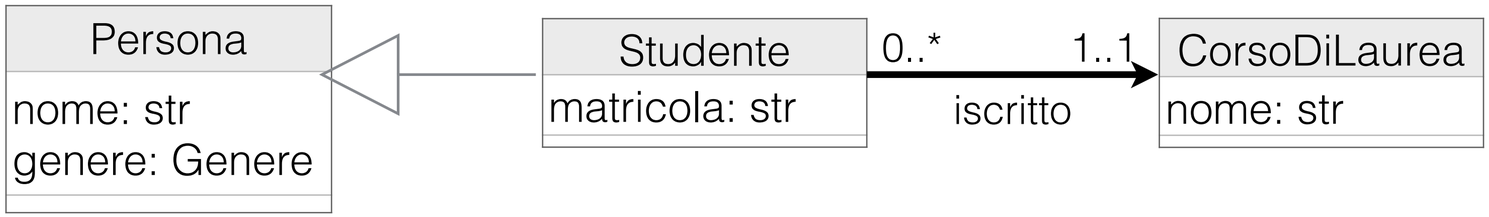


- La classe Studente è coinvolta nell'associazione iscritto con vincolo di molteplicità 1..1
  - La classe Studente deve avere responsabilità nell'associazione iscritto
- Nessuna operazione di classe/use-case chiede che, dato un corso di laurea, si debba poter risalire ai suoi studenti (caso poco calzante in realtà!)
  - La classe CorsoDiLaurea potrebbe non avere responsabilità nell'associazione iscritto

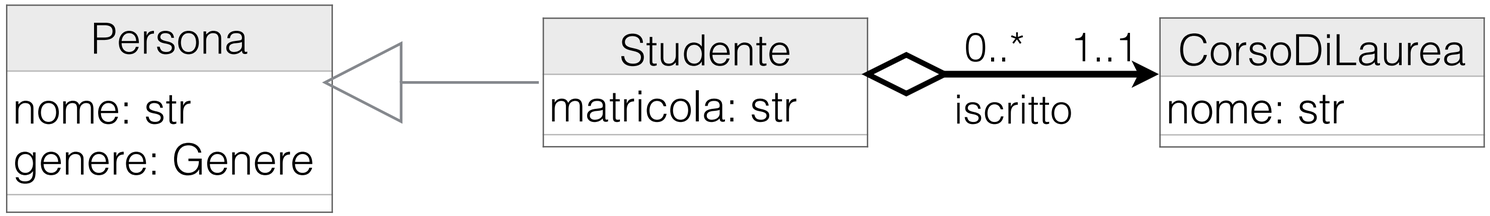
Raccomandazione: essere molto prudenti nel togliere responsabilità di classi ad associazioni. Il codice sarà più semplice ed efficiente, ma meno estendibile

Design di associazioni, responsabilità: diagramma ristrutturato

- Nel diagramma delle classi ristrutturato, alle associazioni a responsabilità di una singola classe si associa un verso di **navigabilità**
  l'associazione può essere "navigata" solo in quel verso



- Una associazione a responsabilità di una singola classe che non ha attributi, può essere ridefinita come **aggregazione UML**



- Una **aggregazione** identifica una relazione "has-a":
  gli oggetti della classe responsabile hanno o possono avere (come "parte", cioè come se fosse un attributo) riferimenti diretti ad uno o più oggetti dell'altra classe (in base al vincolo di molteplicità)

ITC INFORMATION AND COMMUNICATIONS TECHNOLOGY ACADEMY

MODULO: Progettazione
UNITÀ: Progettazione.1

Prof. Toni Mancini
Dipartimento di Informatica
Sapienza Università di Roma

Slide P.1

Design di applicazioni in Python

Implementazione del diagramma UML delle classi ristrutturato

Implementazione del diagramma UML delle classi ristrutturato

- **Obiettivo**: implementare le class del programma.
  - **Input**: diagramma delle classi UML ristrutturato e specifiche ristrutturate
  - **Output**: un insieme di class Python.
- **Metodologia:**
  - Ogni classe UML si traduce in una class Python (possibilmente astratta, se così contrassegnata)
  - Ogni relazione is-a si traduce in una relazione di derivazione tra class in Python
  - Ogni associazione si traduce in una ulteriore class Python e in campi dati delle class Python responsabili
  - Ogni aggregazione si traduce in un campo dati dell'unica class Python responsabile
  - I vincoli di molteplicità dei ruoli di associazione si traducono in codice Python nelle class responsabili
  - I vincoli di identificazione di classe si traducono in oggetti di ulteriori class Python (indici)

ITC INFORMATION AND COMMUNICATIONS TECHNOLOGY ACADEMY

MODULO: Progettazione
UNITÀ: Progettazione.1

Prof. Toni Mancini
Dipartimento di Informatica
Sapienza Università di Roma

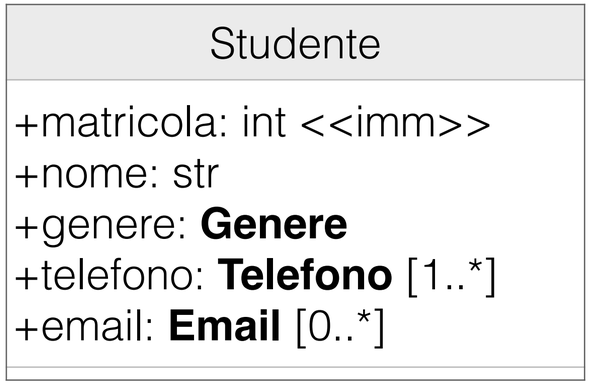
Slide P.1

Progettazione del software
Design di applicazioni in Python

Implementazione del diagramma UML delle classi ristrutturato

Classi

Implementazione del diagramma UML delle classi ristrutturato: tipi di dato

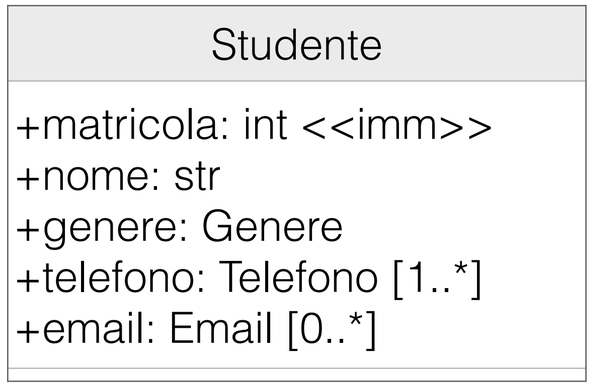


class Genere: …

class Telefono: …

class Email: …

class Modalita: …



Campi dati

class Studente:
campi dati:
_matricola: int #<<imm>>, noto alla nascita
_nome: str # noto alla nascita
_genere: Genere # noto alla nascita
_telefono: **set[Telefono]** # noto alla nascita
_email: **set[Email]** # non noto alla nascita

Implementazione del diagramma UML delle classi ristrutturato: classi

Studente
+matricola: int <<imm>> +nome: str +genere: Genere +telefono: Telefono [1..*] +email: Email [0..*]

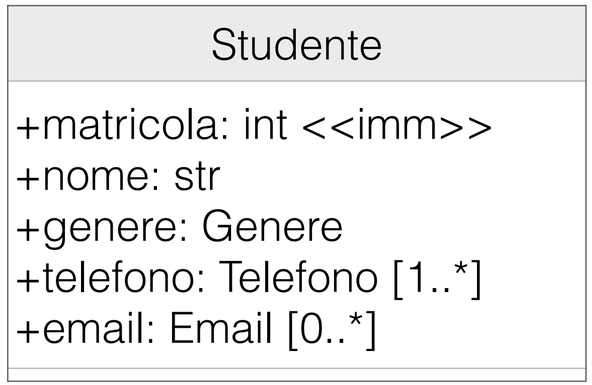
```
class Studente:
  # campi dati:
  _matricola: int # <<imm>>, noto alla nascita
  _nome: str # noto alla nascita
  _genere: Genere # noto alla nascita
  _telefono: set[Telefono] # noto alla nascita
  _email: set[Email] # certamente non noto alla
  nascita
```

Metodi getter:

- un metodo getter per ogni attributo public (+)

```
def matricola(self)->int:
  return self._matricola
def nome(self)->str:
  return self._nome
def genere(self)->Genere:
  return self._genere
def telefono(self)->frozenset[Telefono]:
  return frozenset(self._telefono) # una copia immutabile!
def email(self)->frozenset[Email]:
  return frozenset(self._email) # una copia immutabile!
```

Implementazione del diagramma UML delle classi ristrutturato: classi




```
class Studente:
  # campi dati:
  _matricola: int # <<imm>>, noto alla nascita
  _nome: str # noto alla nascita
  _genere: Genere # noto alla nascita
  _telefono: set[Telefono] # noto alla nascita
  _email: set[Email] # certamente non noto alla
  nascita
```

Metodi setter:

- un metodo setter per ogni attributo public…
- **tranne** per quelli <immutable>> **e** noti alla nascita

```
# def set_matricola(…) <— no, è <<imm>> e noto alla nascita

def set_nome(self, n:str)->None:
  self._nome:str = n
def set_genere(self, g:Genere)->None:
  self._genere:Genere = g
  # nota: g è immutabile

# setter per gli attributi multivalore telefono e email?
```

Implementazione del diagramma UML delle classi ristrutturato: classi

Studente
+matricola: int <<imm>> +nome: str +genere: Genere +telefono: Telefono [1..*] +email: Email [0..*]

```
class Studente:
  # campi dati:
  _matricola: int # <<imm>>, noto alla nascita
  _nome: str # noto alla nascita
  _genere: Genere # noto alla nascita
  _telefono: set[Telefono] # noto alla nascita
  _email: set[Email] # certamente non noto alla
  nascita
```

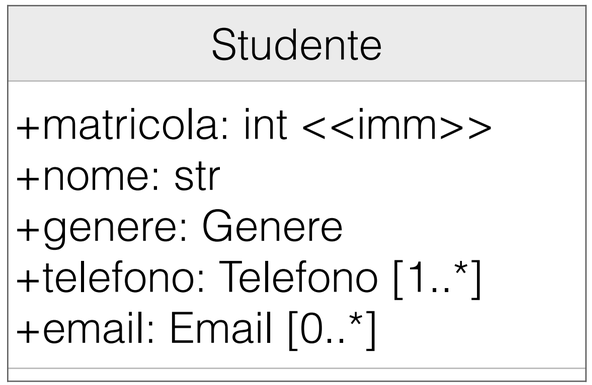
Metodi setter (continua):

- un metodo setter per ogni attributo public…
- **tranne** per quelli <immutable>> **e** noti alla nascita
- per gli **attributi multivalore**: **add_attr(), remove_attr()**

```
def add_telefono(self, t:Telefono)->None:
  self._telefono.add(t)
def remove_telefono(self, t:Telefono)->None:
  self._telefono.remove(t)

def add_email(self, e:Email)->None:
  self._email.add(e)
def remove_email(self, e:Email)->None:
  self._email.remove(e)
```

Implementazione del diagramma UML delle classi ristrutturato: classi




```
class Studente:
  # campi dati:
  _matricola: int # <<imm>>, noto alla nascita
  _nome: str # noto alla nascita
  _genere: Genere # noto alla nascita
  _telefono: set[Telefono] # noto alla nascita
  _email: set[Email] # certamente non noto alla nascita
```

Metodo __init__():

- un arg. per ogni attributo che è o può essere noto alla nascita
- se arg. può essere (ma non è certamente) noto alla nascita, default = None

```
def __init__(self, mat:int, nome:str, g:Genere,
t:Telefono):

    self._telefono = set()
    self._email = set()

    # mat è <<imm>> e noto alla nascita, quindi non ha
    metodo setter:
    # mettiamo il codice qui (unico punto dove può essere
    eseguito)
    self._matricola:int = mat

    self.set_nome(nome)
    self.set_genere(g)
    self.add_telefono(t)
```

Implementazione del diagramma UML delle classi ristrutturato: classi

Corso
+nome: str +modalita: **Modalita** [0..1] <<imm>>

```
class Corso:
  # campi dati:
  _nome: str # noto alla nascita
  _modalita: Modalita # [0..1] <<imm>>,
  può, ma non deve, essere noto alla
  nascita
```

Metodi getter:

- un metodo getter per ogni attributo public (+)

```
def nome(self)->str:
  return self._nome
def modalita(self)->ModalitaINone:
  return self._modalita # può essere None (molt. [0..1])
```

Implementazione del diagramma UML delle classi ristrutturato: classi

Corso
+nome: str +modalita: **Modalita** [0..1] <<imm>>

class Corso:
  # campi dati:
  _nome: str # noto alla nascita
  _modalita: Modalita # [0..1] <<imm>>, può, ma non deve, essere noto alla nascita

Metodi setter:

- un metodo setter per ogni attributo public…
- **tranne** per quelli <immutable>> **e** noti alla nascita

```
def set_nome(self, n:str)->None:
  self._nome = n

# <<imm>>, ma può essere non noto alla nascita
def set_modalita(self, mod:ModalitaINone)->None:
  try:
    if self._modalita:
      raise ValueError("attr. <<imm>> già assegnato")
  except AttributeError:
    pass # self._modalita non è mai stato definito
  self._modalita = mod # può essere None
```

Implementazione del diagramma UML delle classi ristrutturato: classi

Corso
+nome: str +modalita: **Modalita** [0..1] <<imm>>

```
class Corso:
  # campi dati:
  _nome: str # noto alla nascita
  _modalita: Modalita # [0..1] <<imm>>,
  può, ma non deve, essere noto alla
  nascita
```

Metodo __init__():

- un arg. per ogni attributo che è o può essere noto alla nascita
- se arg. può essere (ma non è certamente) noto alla nascita, default = None

```
def __init__(self, nome:str, mod:ModalitaINone = None):

  self.set_nome(nome)
  self.set_modalita(mod)
```

ITC INFORMATION AND COMMUNICATIONS TECHNOLOGY ACADEMY

MODULO: Progettazione
UNITÀ: Progettazione.1

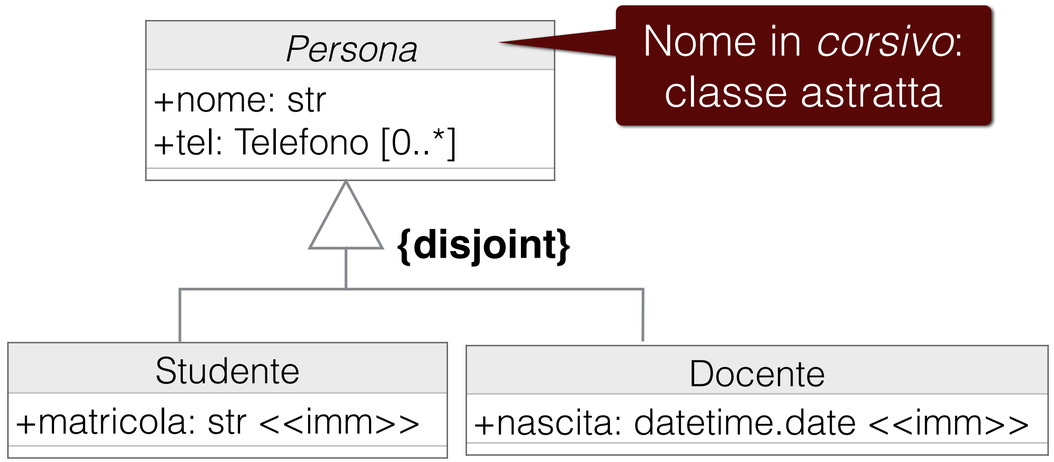
Prof. Toni Mancini
Dipartimento di Informatica
Sapienza Università di Roma

Slide P.1

Progettazione del software
Design di applicazioni in Python

Implementazione del diagramma UML delle classi ristrutturato

Generalizzazioni




```
class Persona(ABC): # class astratta
  # campi dati:
  _nome: str # noto alla nascita
  _telefono: set[Telefono] # può essere non noto alla nascita

  @abstractmethod
  def __init__(self, *, n:str, t: Telefono|None=None): …
  def nome(self)->str: …
  def telefono(self)->frozenset[Telefono]: …
  def set_nome(self, n:str)->None: …
  def add_telefono(self, t:Telefono)->None: …
  def remove_telefono(self, t:Telefono)->None: …
```

Class Python, campi dati e metodi __init__

```
class Studente(Persona): # subclass
  # campi dati:
  _matricola: str, <<imm>>, noto alla nascita
  def matricola(self)->str: …

  def __init__(self, *, n:str, t:Telefono|None=None, mat:str):
    self.super().__init__(n, t) # chiamo __init__() della superclass
    # gestione di mat (<<imm>> e nota alla nascita)
    self._matricola:str = mat
```

```
class Docente(Persona): # subclass
  # campi dati:
  # _nascita: datetime.date, <<imm>>, noto alla nascita
  def nascita(self)->datetime.date: …

  def __init__(self, *, n:str, t:Telefono|None=None,
  nasc:datetime.date):
    … # simile al metodo __init__() di Studente
```

Fai click sui nomi di class sottolineati per accedere al codice completo

ITC INFORMATION AND COMMUNICATIONS TECHNOLOGY ACADEMY

MODULO: Progettazione
UNITÀ: Progettazione.1

Prof. Toni Mancini
Dipartimento di Informatica
Sapienza Università di Roma

Slide P.1

Progettazione del software
Design di applicazioni in Python

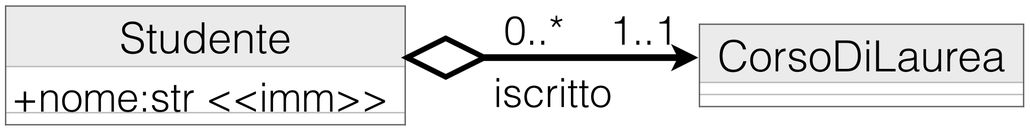
Implementazione del diagramma UML delle classi ristrutturato

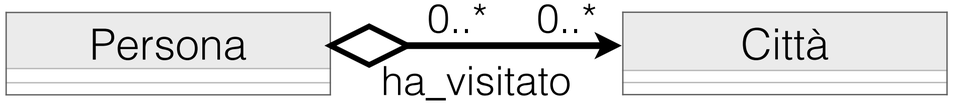
Aggregazioni

Implementazione del diagramma UML delle classi ristrutturato: aggregazioni

Associazioni a responsabilità singola e senza attributi, ridefinite come **aggregazioni** UML

—> Gestite come se fossero attributi dell'unica classe responsabile





class Studente:

```
_nome:str
_iscritto:CorsoDiLaurea
def nome(self)->str: …
def iscritto(self)->CorsoDiLaurea:
  return self._iscritto
def set_nome(self, nome:str)->None: …
def set_iscritto(self, iscritto:CorsoDiLaurea)->None:
  self._iscritto = iscritto
def __init__(self, *, nome:str,
iscritto:CorsoDiLaurea):
  self.set_nome(nome)
  self.set_iscritto(iscritto)
```

class Persona:

```
_ha_visitato:set[Citta] # certamente non noto alla nascita
def ha_visitato(self)->frozenset[Citta]:
  return frozenset(self._ha_visitato)
def add_ha_visitato(self, c:Citta)->None:
  self._ha_visitato.add(c)
def remove_ha_visitato(self, c:Citta)->None:
  if c:
    self._ha_visitato.remove(c)
def __init__(…):
  self._ha_visitato:set[Citta] = PedanticSet()
  …
```

Fai click sui nomi di class sottolineati per accedere al codice completo

ITC INFORMATION AND COMMUNICATIONS TECHNOLOGY ACADEMY

MODULO: Progettazione
UNITÀ: Progettazione.1

Prof. Toni Mancini
Dipartimento di Informatica
Sapienza Università di Roma

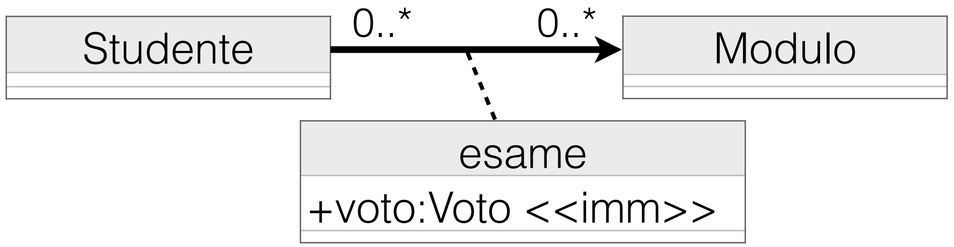
Slide P.1

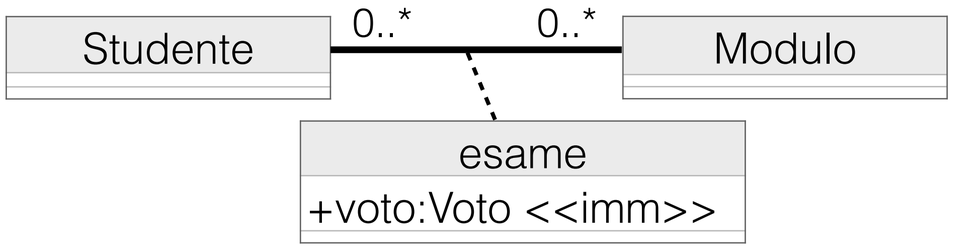
Progettazione del software
Design di applicazioni in Python

Implementazione del diagramma UML delle classi ristrutturato

Associazioni

Associazioni, possibilmente con attributi, a responsabilità qualsiasi





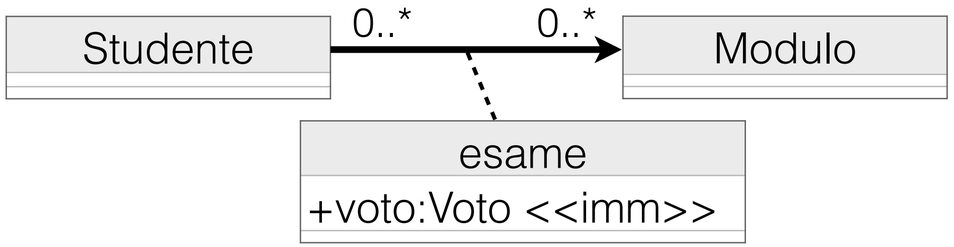
Class Python e campi dati

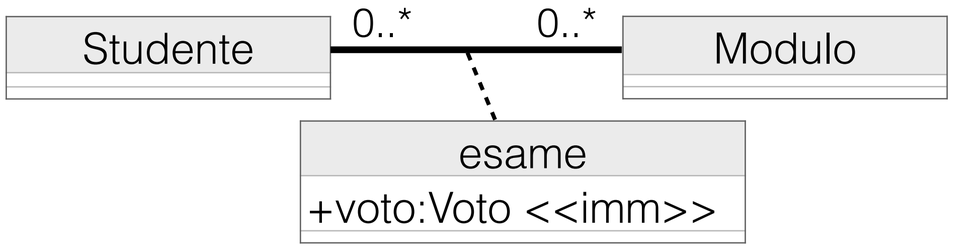
```
class esame:
  # Classe privata le cui istanze rappresentano link
  class _link:
    _studente:Studente # sempre immutabile e noto alla nascita
    _modulo:Modulo # sempre immutabile e noto alla nascita
    _voto:Voto # <<imm>>, noto alla nascita

    # metodi getter
    def studente(self)->Studente:
      return self._studente
    def modulo(self)->Modulo:
      return self._modulo
    def voto(self)->Voto:
      return self._voto
```

Fai click sui nomi di class sottolineati per accedere al codice completo

Associazioni, possibilmente con attributi, a responsabilità qualsiasi





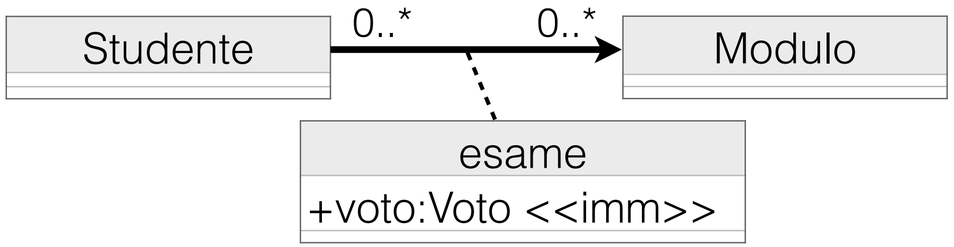
Metodo __init__

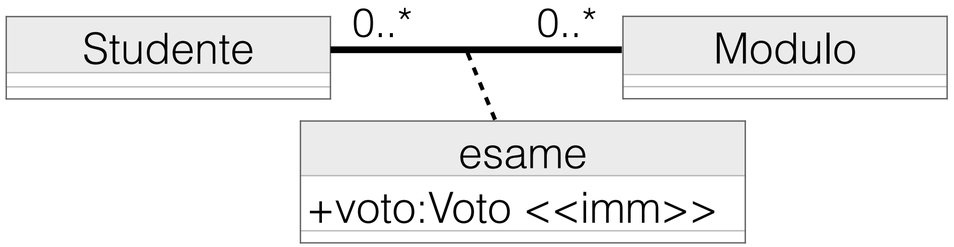
```
class esame:
  # Classe privata le cui istanze rappresentano link
  class _link:
    _studente:Studente # sempre immutabile e noto alla nascita
    _modulo:Modulo # sempre immutabile e noto alla nascita
    _voto:Voto # <<imm>>, noto alla nascita

    def __init__(self, s:Studente, m:Modulo, v:Voto):
      self._studente:Studente = s
      self._modulo:Modulo = m
      self._voto:Voto = v
```

Fai click sui nomi di class sottolineati per accedere al codice completo

Associazioni, possibilmente con attributi, a responsabilità qualsiasi





Metodi __hash__ e __eq__

- necessari per preservare la semantica UML (no link identici)

```
class esame:
  # Classe privata le cui istanze rappresentano link
  class _link:
    _studente:Studente # sempre immutabile e noto alla nascita
    _modulo:Modulo # sempre immutabile e noto alla nascita
    _voto:Voto # <<imm>>, noto alla nascita

    def __hash__(self)->int:
      return hash(  (self.studente(), self.modulo())  )
    def __eq__(self, other:Any)->bool:
      if type(self) != type(other) \ # corretto anche se other is None
              or hash(self) != hash(other):
        return False
      return ( self.studente(), self.modulo() ) == \
              (other.studente(), other.modulo() )
```

Fai click sui nomi di class sottolineati per accedere al codice completo

ITC INFORMATION AND COMMUNICATIONS TECHNOLOGY ACADEMY

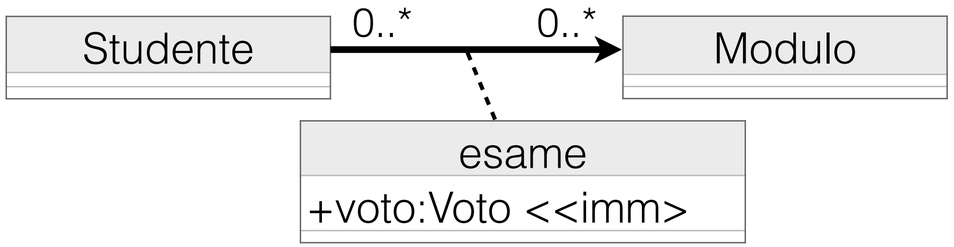
MODULO: Progettazione
UNITÀ: Progettazione.1

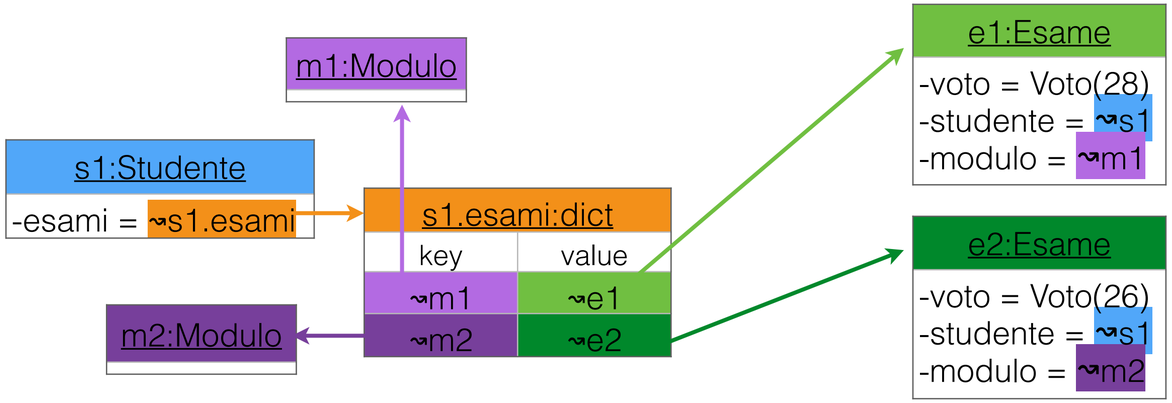
Prof. Toni Mancini
Dipartimento di Informatica
Sapienza Università di Roma

Slide P.1

Progettazione del software
Design di applicazioni in Python
Implementazione del diagramma UML delle classi ristrutturato
Associazioni
Associazioni a resp. singola

Implementazione della **responsabilità singola**





Fai click sui nomi di class sottolineati per accedere al codice completo

Class Python, campi dati, getter e setter

```
class Studente:
    _nome:str
    _esami:dict[Modulo,esame._link] # mutabile, non noto alla nascita
    def nome(self)->str:
        return self._nome
    def esami(self)->frozenset[weakref[esame._link]]:
        return frozenset([weakref.ref(l) for l in self._esami.values()])
    def esame(self, modulo:Modulo)->weakref[esame._link]:
        return weakref.ref(self._esami[modulo])
    def __init__(self, nome:str):
        self._nome = nome
        self._esami:dict[Modulo,esame._link] = dict()
    def add_link_esame(self, modulo:Modulo, voto:Voto)->None:
        l = esame._link(self, modulo, voto)
        if modulo in self._esami:
          raise KeyError(f"Duplicate link ({self}, {modulo}) not allowed")
        self._esami[modulo] = l
    def remove_link_esame(self, l:weakref[esame._link])->None:
        if l().studente() is not self:
            raise ValueError("link does not involve me")
        del self._esami[ l().modulo() ]
```

ITC INFORMATION AND COMMUNICATIONS TECHNOLOGY ACADEMY

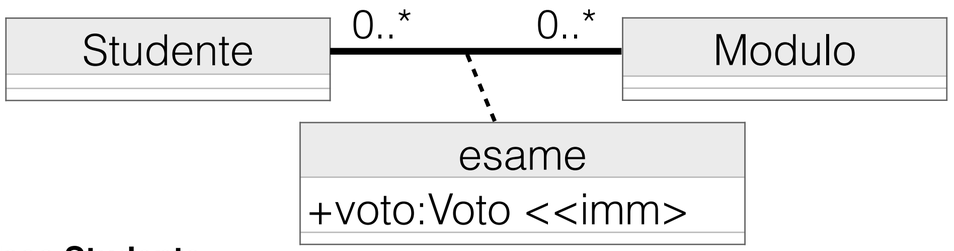
MODULO: Progettazione
UNITÀ: Progettazione.1

Prof. Toni Mancini
Dipartimento di Informatica
Sapienza Università di Roma

Slide P.1

Progettazione del software
Design di applicazioni in Python
Implementazione del diagramma UML delle classi ristrutturato
Associazioni
Associazioni a resp. doppia

Implementazione della **responsabilità doppia**



class Studente:

```
_nome:str
_esami:dict[Modulo,esame.link] # mutabile, non noto alla nascita
def nome(self)->str:
    return self._nome
def esami(self)->frozenset[weakref[esame._link]]:
    return frozenset([weakref.ref(l) for l in self._esami.values()])
def esame(self, modulo:Modulo)->weakref[esame._link]:
    return weakref.ref(self._esami[modulo])
def __init__(self, nome:str):
    self._nome = nome
    self._esami:dict[Modulo,esame._link] = dict()
def _add_link_esame(self, l:esame._link)->None:
    if l.studente() is not self:
        raise ValueError("link does not involve me")
    if l.modulo() in self._esami:
        raise KeyError(f"Duplicate link ({self}, {l.modulo()}) not allowed")
    self._esami[l.modulo()] = l
def _remove_link_esame(self, l:esame._link)->None:
    if l.studente() is not self:
        raise ValueError("link does not involve me")
    del self._esami[ l.modulo() ]
```

Class, campi dati, metodi getter e setter

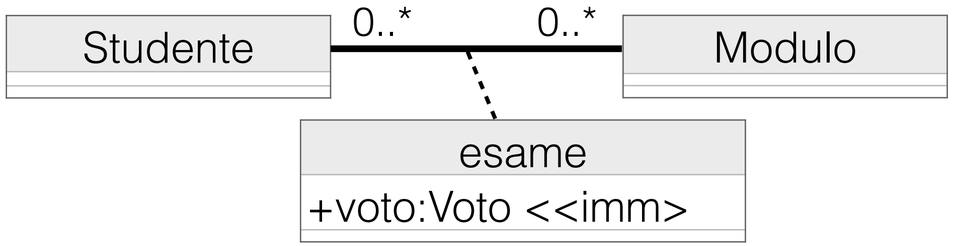
- gestione simmetrica, ma con metodi setter privati che supportano (ma non da soli!) la creazione dei link (segue…)

class Modulo:

```
_nome:str # immutabile
_esami:dict[Studente,esame.link] # mutabile, non noto alla nascita
def nome(self)->str:
    return self._nome
def esami(self)->frozenset[weakref[esame._link]]:
    return frozenset([weakref.ref(l) for l in self._esami.values()])
def esame(self, studente:Studente)->weakref[esame._link]:
    return weakref.ref(self._esami[studente])
def __init__(self, nome:str):
    self._nome = nome
    self._esami:dict[Studente,esame._link] = dict()
def _add_link_esame(self, l:esame._link)->None:
    if l.modulo() is not self:
        raise ValueError("link does not involve me")
    if l.studente() in self._esami:
        raise KeyError(f"Duplicate link ({l.studente()}, {self}) not allowed")
    self._esami[l.studente()] = l
def _remove_link_esame(self, l:esame._link)->None:
    if l.modulo() is not self:
        raise ValueError("link does not involve me")
    del self._esami[ l.studente() ]
```

Fai click sui nomi di class sottolineati per accedere al codice completo

Implementazione della **responsabilità doppia**



Creazione e rimozione di link

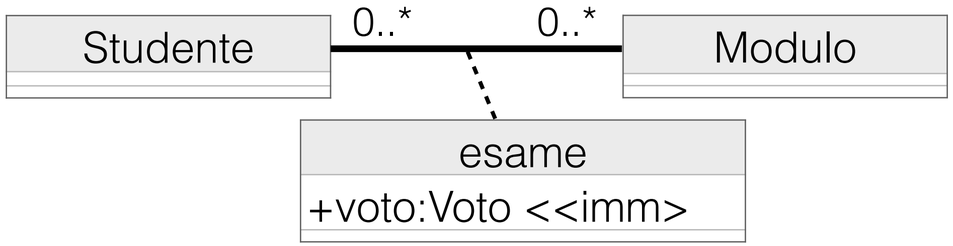
- creazione dei link centralizzata in @classmethod della class che implementa l'associazione (pattern "factory")

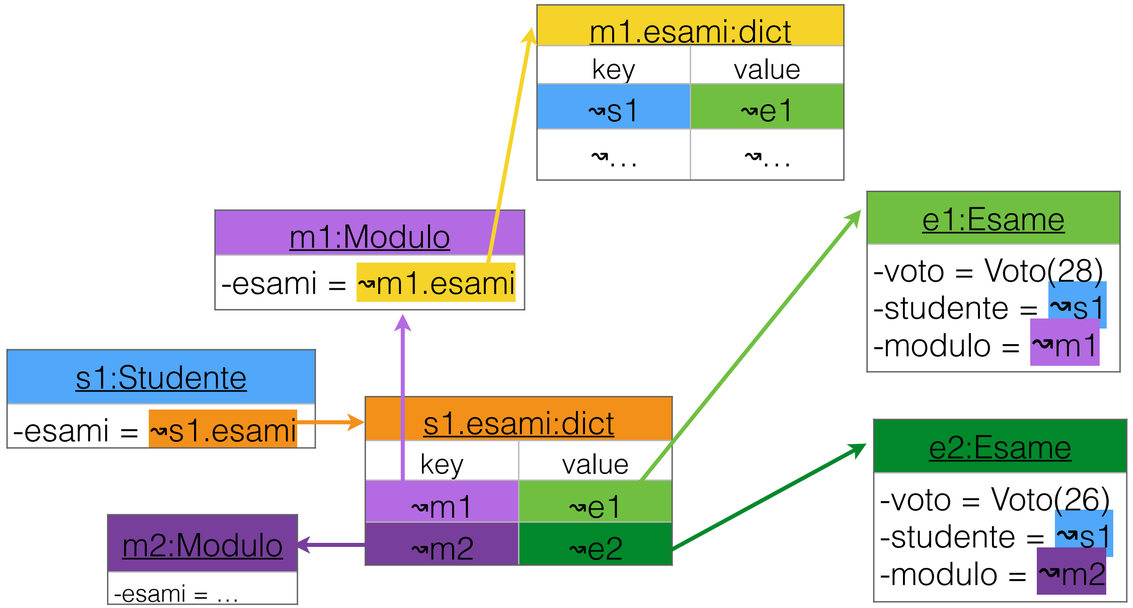
```
class esame:
    # Associazione a resp. doppia
    # -> Factory per la creazione e rimozione di link di associazione esame
    @classmethod
    def add(cls, s:Studente, m:Modulo, v:Voto)->None:
        l = esame._link(s,m,v)
        l.studente()._add_link_esame(l)
        l.modulo()._add_link_esame(l)
    @classmethod
    def remove(cls, l:weakref[esame._link])->None:
        if l() is None:
            raise ValueError("l cannot be None")
        l().studente()._remove_link_esame(l())
        l().modulo()._remove_link_esame(l())
        del l

    # Classe privata le cui istanze rappresentano link
    class _link:
        …
```

Fai click sui nomi di class sottolineati per accedere al codice completo

Implementazione della **responsabilità doppia**





Creazione e rimozione di link “asimmetrica”

- In alcuni casi, può essere opportuno che
  - sebbene l’associazione sia a responsabilità doppia
  - una delle due classi sia deputata a creare e rimuovere i link

—> i @classmethod per creare e rimuovere i link si spostano in quella classe (gestione asimmetrica dell’associazione a resp. doppia)

class Studente:

```
…
# I metodi setter per l’associazione diventano pubblici per una delle due class
def add_link_esame(self, m:Modulo, v:Voto)->None:
    l = esame._link(self,m,v) # controlla che m e v non siano None
    self._esami.add( l )
    m._add_link_esame( l ) # l’altra class non cambia
def remove_link_esame(self, l:weakref[esame._link])->None:
    if l().studente() is not self:
      raise ValueError("link does not involve me")
    del self._esami[ l().modulo() ]
    l().modulo()._remove_link_esame(l())
    del l
```

class esame:

…

ITC INFORMATION AND COMMUNICATIONS TECHNOLOGY ACADEMY

MODULO: Progettazione
UNITÀ: Progettazione.1

Prof. Toni Mancini
Dipartimento di Informatica
Sapienza Università di Roma

Slide P.1

Progettazione del software
Design di applicazioni in Python

Implementazione del diagramma UML delle classi ristrutturato

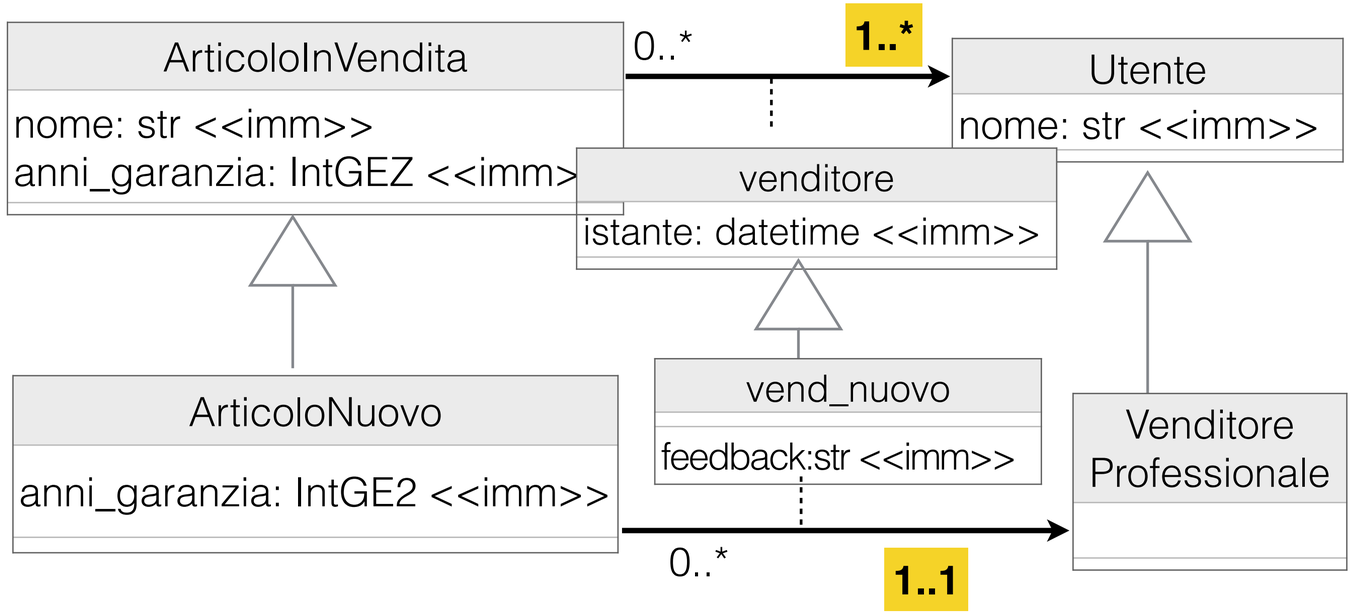
Generalizzazioni tra associazioni

Implementazione di generalizzazioni tra associazioni

Implementazione della **responsabilità singola**

Class Python e campi dati

```
class venditore:
    class _link:
      _articolo:Articolo # <<imm>> e noto alla nascita
      _utente:Utente # <<imm>> e noto alla nascita
      _istante:datetime # <<imm>> e noto alla nascita
      def __init__(self, a:Articolo, u:Utente,
      i:datetime): ...
      def articolo(self)->Articolo: ...
      def utente(self)->Utente: ...
      def istante(self)->datetime: ...
      def __hash__(self): ...
      def __eq__(self, other): ...
```

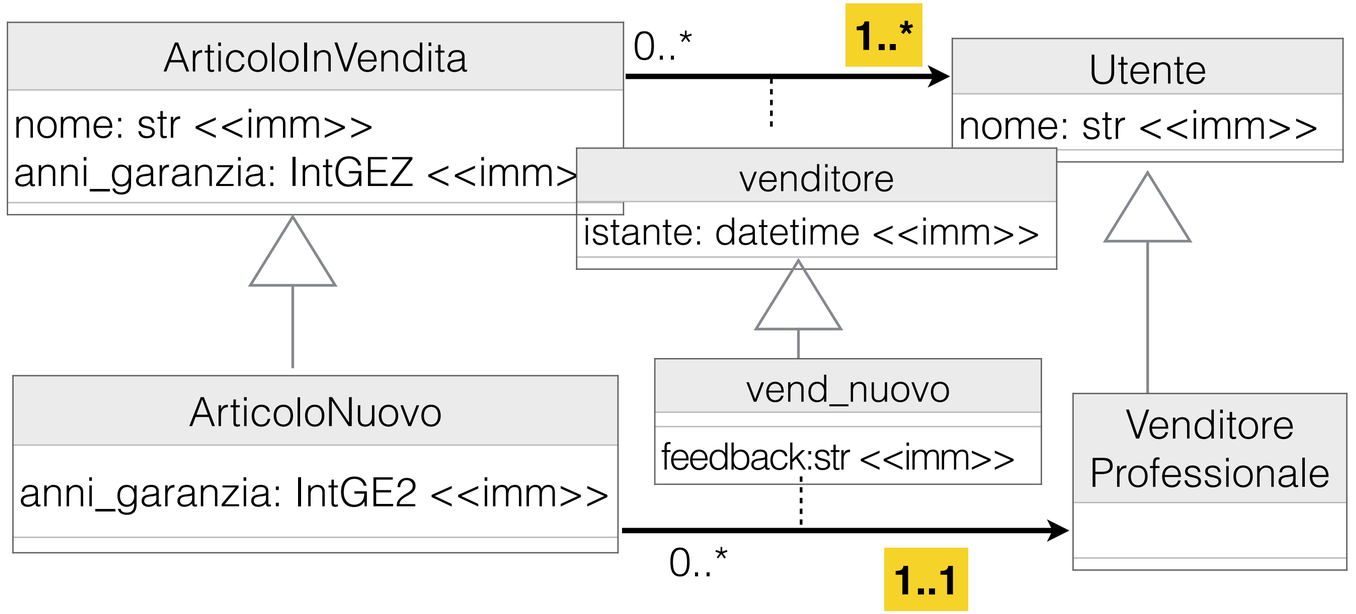



```
class vend_nuovo(venditore):
    class _link(venditore._link):
      _feeedback:str # mutabile, noto alla nascita
      def __init__(self, a:ArticoloNuovo, u:VenditoreProf,
      i:datetime, f:str):
          super().__init__(a, u, i)
          self.set_feedback(f)
      def utente(self)->VenditoreProf:
          return super().utente()
      def feedback(self)->str: ...
      def set_feedback(self, feedback)->None: ...
```

Implementazione della **responsabilità singola**

Class Python e campi dati

```
class Articolo:
    _nome:str # <<imm>>
    _anni_garanzia:IntGEZ # <<imm>>
    _venditori:dict[Utente, venditore._link] # [1..*] mutabile, noto alla nascita
    def nome(self)->str: …
    def garanzia(self)->IntGEZ: …
    def venditori(self)->frozenset[venditore._link]: …
    def venditore(self, utente:Utente)->venditore._linkINone: …
    # gestione associazione venditore a resp. singola
    # metodo protetto, usato anche dalla subclass
    def _add_venditore(self, l:venditore._link)->None:
        if l is None:
            raise ValueError("l cannot be None")
        if l.articolo() is not self:
            raise ValueError("l does not involve me")
        if l.utente() in self._venditori:
            raise ValueError("No duplicate links allowed")
        self._venditori[l.utente()] = l
    # metodo pubblico
    def add_venditore(self, utente:Utente, istante:datetime)->None:
        self._add_venditore(venditore._link(self, utente, istante))
    def remove_venditore(self, l:venditore._link)->None:
        if not l: raise ValueError("l cannot be None")
        if l.articolo() is not self: raise ValueError("l does not involve me")
        try:
            if self._venditori[l.utente()] is not l:  raise RuntimeError("link provided is not stored")
            del self._venditori[l.utente()]
        except KeyError: raise ValueError("link is not stored")
    def _init_nome_gar(self, *, nome:str, garanzia:IntGEZlint):
        if nome is None: raise ValueError("nome cannot be None")
        self._nome = nome
        if garanzia is None: raise ValueError("garanzia cannot be None")
        self._garanzia = IntGEZ(garanzia) if not isinstance(garanzia, IntGEZ) else garanzia
        self._venditori:dict[Utente, venditore._link] = dict()
    def __init__(self, *, nome:str, garanzia:IntGEZ, utente:Utente, istante:datetime):
            self._init_nome_gar(nome=nome, garanzia=garanzia)
            self.add_venditore(utente, istante)
```




```
class ArticoloNuovo(Articolo):
    _anni_garanzia:IntGE2 # <<imm>>
    _vend_nuovo:vend_nuovo._link
    def __init__(self, *, nome:str, garanzia:IntGE2lint, utente:VenditoreProf, istante:datetime, feedback:str):
        # Nota: non chiamo super(), perché devo chiamare ArticoloNuovo.set_venditore() che ha segnatura diversa
        self._init_nome_gar(nome=nome, garanzia=IntGE2(garanzia))
        self._vend_nuovo = None
        self.set_vend_nuovo(utente, istante, feedback)
    def vend_nuovo(self)->vend_nuovo._linkINone: return self._vend_nuovo
    def set_vend_nuovo(self, utente:VenditoreProf, istante:datetime, feedback:str)->None:
        if self._vend_nuovo:
            # remove old link
            super().remove_venditore(self._vend_nuovo)
        self._vend_nuovo = vend_nuovo._link(self, utente, istante, feedback)
        super()._add_venditore(self._vend_nuovo)
    def remove_venditore(self, l:venditore._link)->None:
        if isinstance(l, vend_nuovo._link):
            raise ValueError("Cannot remove link of assoc. venditore which is also of assoc. vend_nuovo")
        super().remove_venditore(l)
```

ITC INFORMATION AND COMMUNICATIONS TECHNOLOGY ACADEMY

MODULO: Progettazione
UNITÀ: Progettazione.1

Prof. Toni Mancini
Dipartimento di Informatica
Sapienza Università di Roma

Slide P.1

Progettazione del software
Design di applicazioni in Python

Implementazione del diagramma UML delle classi ristrutturato

Vincoli di integrità

Implementazione dei vincoli di integrità

Distinguiamo tra:

Vincoli di istanza

Si valutano indipendentemente su ogni singola istanza di una classe o associazione, ad es:

Per ogni p:Persona deve essere: p.is_studente = True se e solo se p.matricola è valorizzato

—> Verificarli nel metodo __init__() e in ogni metodo set() che potrebbe portare a violarli

Vincoli di classe

Si valutano sull'insieme delle istanze di una classe o associazione, ad es:

Vincoli di identificazione di classe (non esistono due oggetti di classe Persona con lo stesso valore per il codice fiscale)

—> Per verificarli, dobbiamo poter risalire a tutti gli oggetti di una classe

Vincoli globali

Si valutano su istanze di classi ed associazioni multiple

ITC INFORMATION AND COMMUNICATIONS TECHNOLOGY ACADEMY

MODULO: Progettazione
UNITÀ: Progettazione.1

Prof. Toni Mancini
Dipartimento di Informatica
Sapienza Università di Roma

Slide P.1

Progettazione del software
Design di applicazioni in Python

Implementazione del diagramma UML delle classi ristrutturato

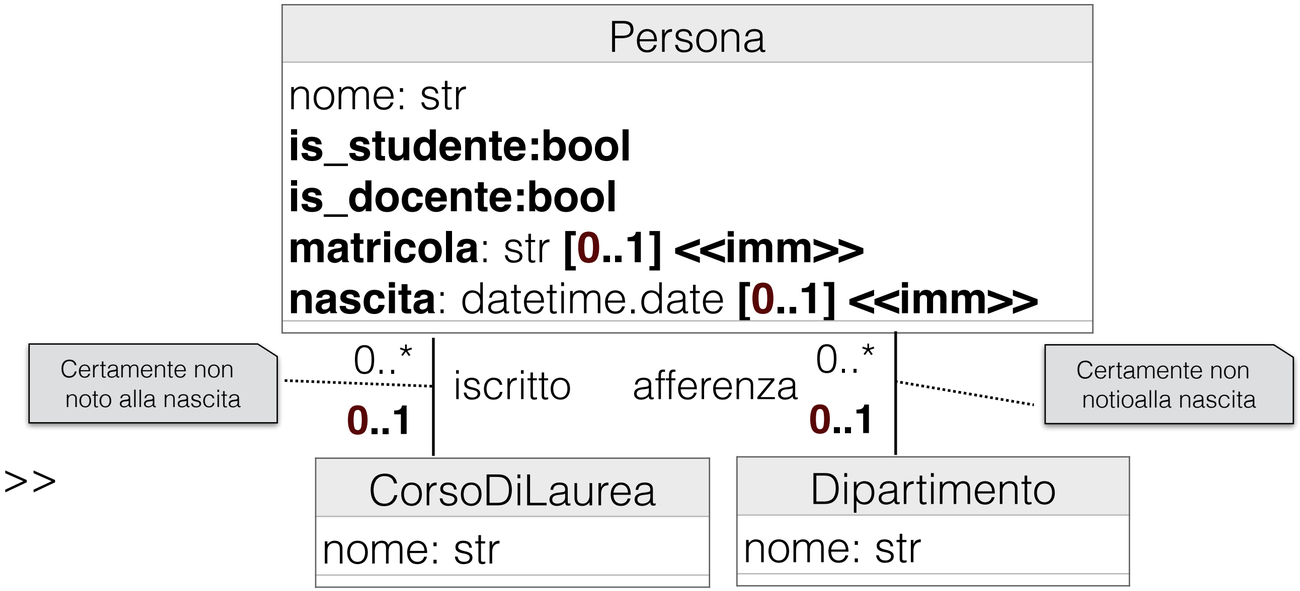
Vincoli di integrità

Vincoli di istanza

Implementazione dei vincoli di integrità: vincoli di istanza

Class Persona:

```
…
def set_dati_studente(self, *, matricola:strlNone=None,
cdl:CorsoDiLaurea)->None:
    if not self._matricola: # self._matricola never set
        if not matricola:
            raise ValueError("matricola cannot be None")
    elif matricola and matricola != self._matricola:
        raise ValueError(f"cannot change already set <<imm>>
          matricola (from {self._matricola} to {matricola})
          for studente {self}")
    if not cdl:
        raise ValueError("cdl cannot be None")
    if matricola:
        self._matricola = matricola
    self._iscritto = cdl
    self._is_studente = True
…
```



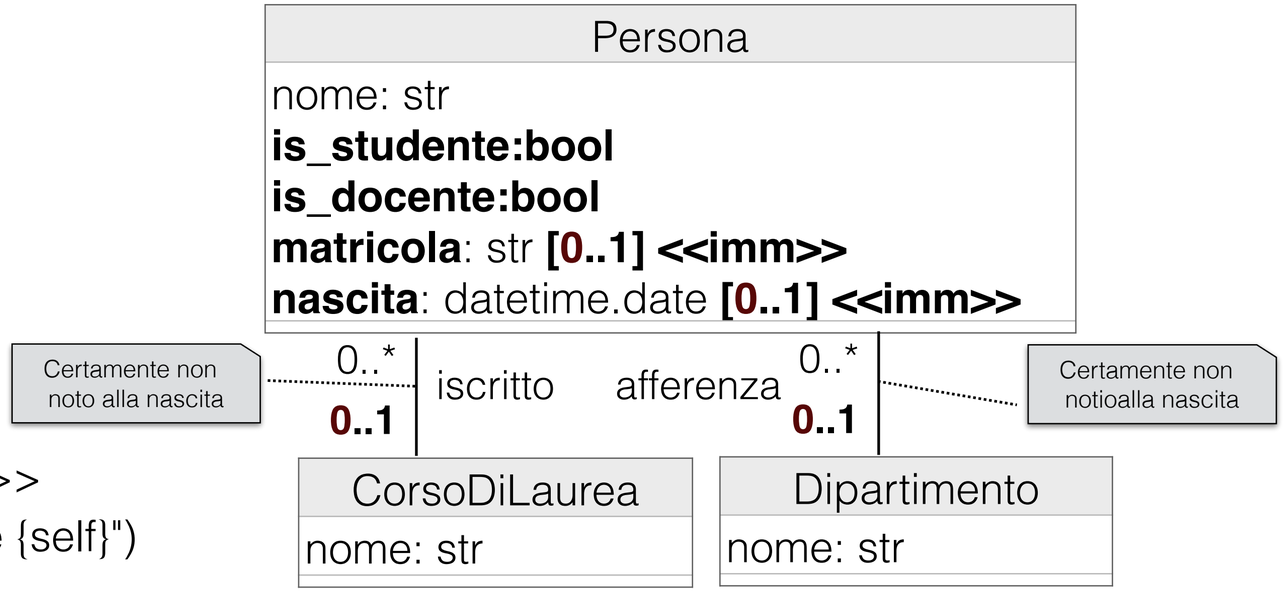
Vincoli esterni: per ogni istanza p di Persona:

1. p.is_studente = TRUE se e solo se p.matricola è valorizzato
2. p.is_studente = TRUE se e solo se p è coinvolto in un link “iscritto”
3. p.is_docente = TRUE se e solo se p.nascita è valorizzato
4. p.is_docente = TRUE se e solo se p è coinvolto in un link “afferenza”
5. p.is_docente = TRUE **oppure** p.is_studente = TRUE

Implementazione dei vincoli di integrità: vincoli di istanza

Class Persona:

```
…
def set_dati_docente(self, *, nascita:datetime.date=None,
dip:Dipartimento)->None:
    if not self._nascita: # self._nascita never set
            if not nascita:
                    raise ValueError("nascita cannot be None")
    elif nascita and nascita != self._nascita:
            raise ValueError(f"cannot change already set <<imm>>
             nascita (from {self._nascita} to {nascita}) for docente {self}")
    if not dip:
            raise ValueError("dip cannot be None")
    if nascita:
            self._nascita = nascita
    self._afferenza = dip
    self._is_docente = True
…
```



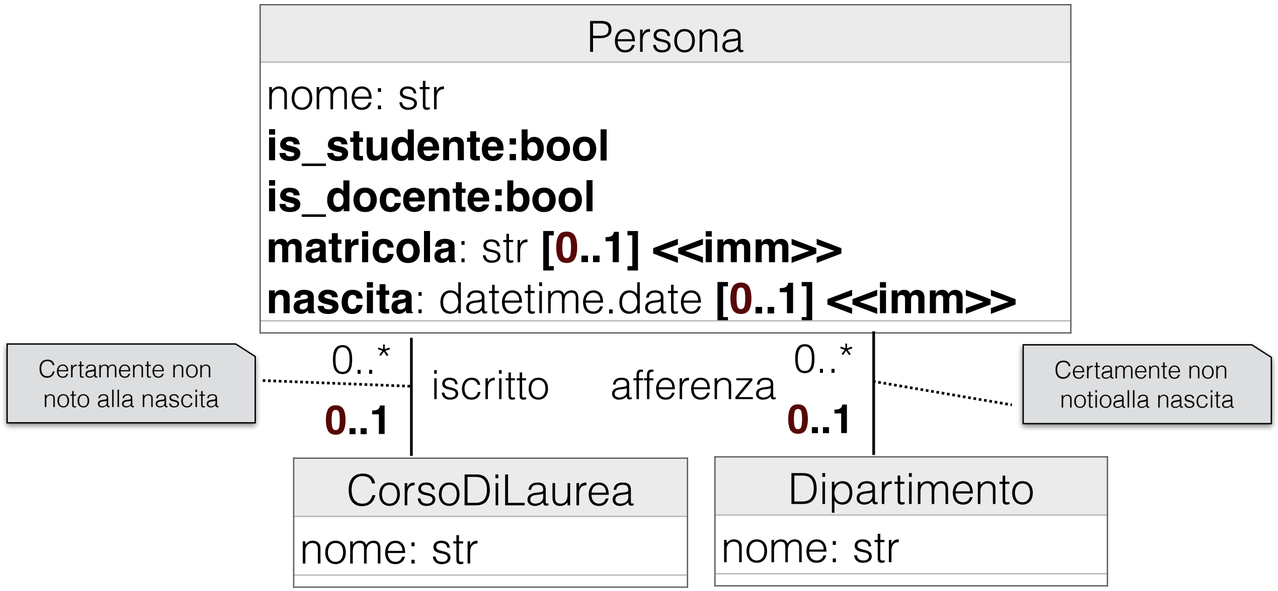
Vincoli esterni: per ogni istanza p di Persona:

1. p.is_studente = TRUE se e solo se p.matricola è valorizzato
2. p.is_studente = TRUE se e solo se p è coinvolto in un link “iscritto”
3. p.is_docente = TRUE se e solo se p.nascita è valorizzato
4. p.is_docente = TRUE se e solo se p è coinvolto in un link “afferenza”
5. p.is_docente = TRUE **oppure** p.is_studente = TRUE

Implementazione dei vincoli di integrità: vincoli di istanza

Class Persona:

```
…
def __init__(self, *, nome:str, matricola:strlNone=None, \
             cdl:CorsoDiLaurealNone=None, \
              nascita:datetime.datelNone=None,
             dip:DipartimentolNone=None):
    self.set_nome(nome)
    self._is_studente = self._is_docente = False
    self._matricola = self._iscritto = self._nascita = \
              self._afferenza = None
    if matricola:
             self.set_dati_studente(matricola=matricola, cdl=cdl)
    if nascita:
             self.set_dati_docente(nascita=nascita, dip=dip)
     if not self.is_studente() and not self.is_docente():
       raise ValueError("[C.5] not satisfied")
def is_studente(self)->bool:
   return self._is_studente
def is_docente(self)->bool:
   return self._is_docente
```



Vincoli esterni: per ogni istanza p di Persona:

1. p.is_studente = TRUE se e solo se p.matricola è valorizzato
2. p.is_studente = TRUE se e solo se p è coinvolto in un link "iscritto"
3. p.is_docente = TRUE se e solo se p.nascita è valorizzato
4. p.is_docente = TRUE se e solo se p è coinvolto in un link "afferenza"
5. p.is_docente = TRUE **oppure** p.is_studente = TRUE

Implementazione dei vincoli di integrità: vincoli di istanza

Esempio di utilizzo:

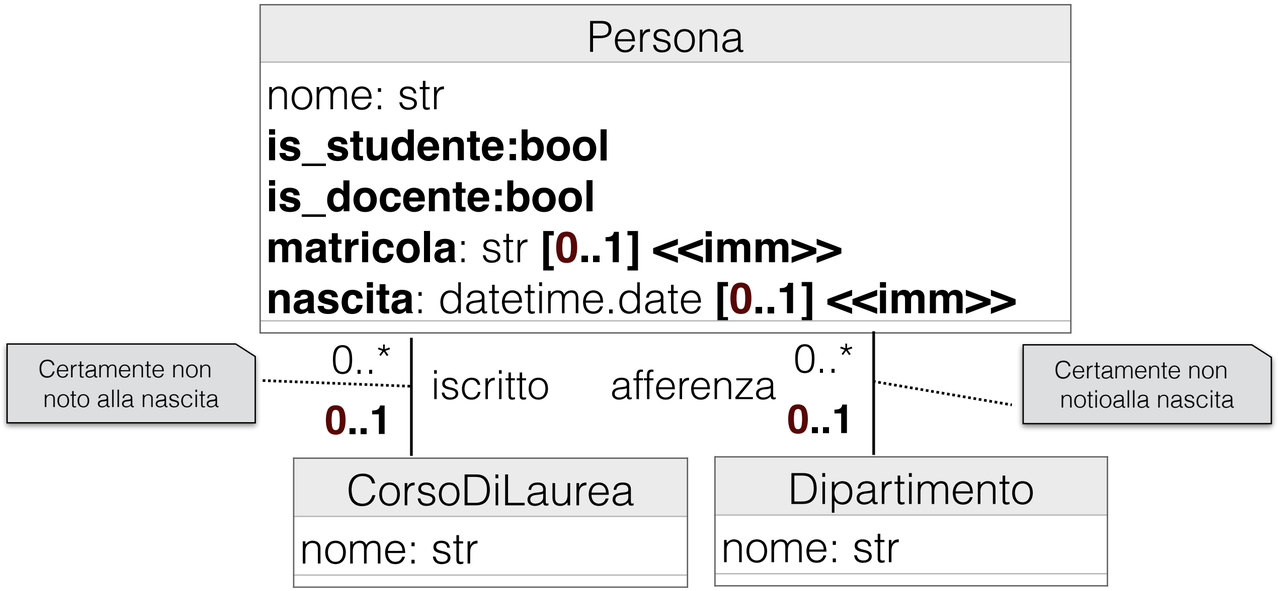
```
cdl = CorsoDiLaurea("Informatica applicata")
cdl2 = CorsoDiLaurea("Informatica teorica")
dip = Dipartimento("Dip. Informatica")
dip2 = Dipartimento("Dip. Matematica")

p = Persona(nome="Mario", matricola="MMM", cdl=cdl)
# p è solo studente

p.set_dati_studente(cdl=cdl2)
# cambio cdl per p in qualità di studente (matricola <<imm>>, invariata)

p.set_dati_docente(nascita=datetime.date.fromisoformat("2000-01-01"), dip=dip)
# p ora è sia studente che docente

p.set_dati_docente(dip=dip2)
# cambio dipartimento di p in qualità di docente (data nascita <<imm>> invariata)
```



Vincoli esterni: per ogni istanza p di Persona:

1. p.is_studente = TRUE se e solo se p.matricola è valorizzato
2. p.is_studente = TRUE se e solo se p è coinvolto in un link “iscritto”
3. p.is_docente = TRUE se e solo se p.nascita è valorizzato
4. p.is_docente = TRUE se e solo se p è coinvolto in un link “afferenza”
5. p.is_docente = TRUE **oppure** p.is_studente = TRUE

ITC INFORMATION AND COMMUNICATIONS TECHNOLOGY ACADEMY

MODULO: Progettazione
UNITÀ: Progettazione.1

Prof. Toni Mancini
Dipartimento di Informatica
Sapienza Università di Roma

Slide P.1

Progettazione del software
Design di applicazioni in Python

Implementazione del diagramma UML delle classi ristrutturato

Vincoli di integrità

Vincoli di classe

Implementazione dei vincoli di integrità: vincoli di classe

```
class Persona:
  _index:Index[str,Self] = Index[str,Self]('Persona')
  @classmethod
  def all(cls):
    return cls._index.all()
  @classmethod
  def get(cls, key:Any)->SelfINone:
    return cls._index.get(key)

  def set_matricola(self, matricola:str)->None:
    if not matricola:
      raise ValueError("matricola cannot be None")
    try:
      self._index.remove(self._matricola)
    except AttributeError: # self._matricola not yet set -> not yet in index
       pass
    self._matricola = matricola
    self._index.add(matricola, self) # Verifica anche {id}

  def __init__(self, *, nome:str, matricola:str):
    if not nome:
      raise ValueError("nome cannot be None")
    self._nome = nome
    self.set_matricola(matricola)
```

Persona
nome: str **<<imm>>** **matricola**: str **{id}**

Implementazione dei vincoli di integrità: vincoli di classe

```
class Persona:
  _index:Index = Index('Persona')
  @classmethod
  def all(cls):
    return cls._index.all()
  @classmethod
  def get(cls, key:Any)->SelfINone:
    return cls._index.get(key)
  def set_matricola(self, matricola:str)->None:
    if not matricola:
      raise ValueError("matricola cannot be None")
    try:
      self._index.remove(self._matricola)
    except AttributeError: # self._matricola not yet set -> not yet in index
      pass
    self._matricola = matricola
    self._index.add(matricola, self) # Verifica anche {id}
  def __init__(self, *, nome:str, matricola:str):
    if not nome:
      raise ValueError("nome cannot be None")
    self._nome = nome
    self.set_matricola(matricola)
```

Persona
nome: str **<<imm>>** **matricola**: str **{id}**

class Index —> conserva un riferimento debole a tutti gli oggetti della classe

Implementazione dei vincoli di integrità: vincoli di classe -> Class Index

```
from typing import *
from weakref import WeakValueDictionary, ReferenceType
KeyType = TypeVar('KeyType')
ValueType = TypeVar('ValueType')
class Index(Generic[KeyType, ValueType]):
    _objects: WeakValueDictionary[KeyType, ValueType]

    def __init__(self, name:str):
        self._name:str = name
        self._objects:WeakValueDictionary[KeyType, ValueType] = WeakValueDictionary()

    def __str__(self)->str:
        return (f"Index {self.name()}:\n - length: {len(self._objects)}\n - keys = [{self._objects.keys()}]")

    def name(self)->str:
        return self._name

    def add(self, _id:KeyType, obj:ValueType)->None:
        if _id in self._objects:
            raise KeyError(f"Duplicate key {_id} for class {type(obj)}")
        self._objects[_id] = obj

    def remove(self, _id:KeyType)->None:
        if _id is not None:
            del self._objects[_id]

    def get(self, _id:KeyType)->ValueTypeINone:
        return self._objects.get(_id, None)

    def all(self)->Generator[ValueType, None, None]:
        return self._objects.values()
```

Dizionario:
- Chiave: tupla di valori degli attributi/ruoli {id}
- Valore: riferimento *debole* all'oggetto con quei valori

Riferimento debole ad oggetto (package weakref)
Riferimento ad oggetto che permette la rimozione automatica dell'oggetto dalla memoria quando gli unici riferimenti all'oggetto sono deboli

Aggiunge la coppia _id -> obj nel dizionario,dove _id è il valore (o tupla di valori) degli attributi {id} di obj. Genera KeyError se il dizionario ha già una coppia con chiave _id

Rimuove dall'indice la coppia con chiave _id

Restituisce l'oggetto con chiave _id (se esiste)

Restituisce un generatore di tutti gli oggetti della classe

Implementazione dei vincoli di integrità: vincoli di classe

```
class Persona:
  _index:Index = Index('Persona')
  @classmethod
  def all(cls):
    return cls._index.all()
  @classmethod
  def get(cls, key:Any)->SelfINone:
    return cls._index.get(key)

  def set_matricola(self, matricola:str)->None:
    if not matricola:
      raise ValueError("matricola cannot be None")
    try:
      self._index.remove(self._matricola)
    except AttributeError: # self._matricola not yet set -> not yet in index
       pass
    self._matricola = matricola
    self._index.add(matricola, self) # Verifica anche {id}

  def __init__(self, *, nome:str, matricola:str):
    if not nome:
      raise ValueError("nome cannot be None")
    self._nome = nome
    self.set_matricola(matricola)
```

Persona
nome: str **<<imm>>** **matricola**: str **{id}**

Accesso controllato all'indice

Restituisce un iterable su tutti gli oggetti della classe

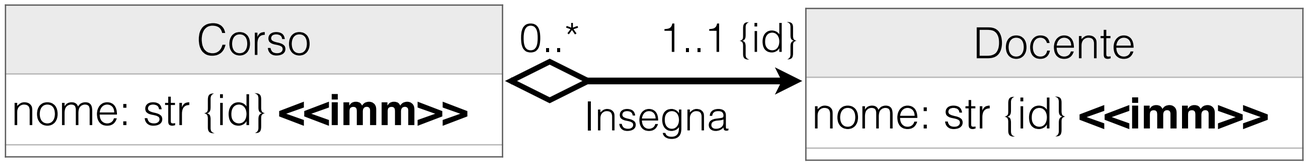
Restituisce l'oggetto dato il valore degli attributi {id}

Rimuove l'oggetto dall'indice (indicizzato con il vecchio valore per gli attributi {id})

Inserisce l'oggetto nell'indice (indicizzato con il nuovo valore per gli attributi {id})

Il metodo set_matricola() si occupa anche di inserire self nell'indice

Implementazione dei vincoli di integrità: vincoli di classe




```
class Corso:
  _index:Index[tuple[str,Docente],Self] = Index[tuple[str,Docente],Self]('Corso')
  @classmethod
  def all(cls)->Generator[Self, None, None]:
    return cls._index.all()
  @classmethod
  def get(cls, key:tuple[str,Docente])->Self|None:
    return cls._index.get(key)

  # _nome: str <<imm>>, noto alla nascita {id}
  # _docente: Docente <<imm>>, noto alla nascita {id}
  def nome(self)->str:
    return self._nome
  def docente(self)->Docente:
    return self._docente
  def set_docente(self, docente:Docente)->None:
    if not docente:
      raise ValueError("docente cannot be None")
    try:
      self._index.remove( (self._nome, self._docente) )
    except AttributeError: # self._docente not yet set -> not yet in index
      pass
    self._docente = docente
    self._index.add( (self._nome, self._docente), self) # Verifica anche {id}
  def __init__(self, *, nome:str, docente:Docente):
    if not nome:
      raise ValueError("nome cannot be None")
    self._nome = nome
    self.set_docente(docente)
```

Come nel caso precedente, ma ora la chiave nell'indice 'Corso' è la tupla (nome, docente)

ITC INFORMATION AND COMMUNICATIONS TECHNOLOGY ACADEMY

MODULO: Progettazione
UNITÀ: Progettazione.1

Prof. Toni Mancini
Dipartimento di Informatica
Sapienza Università di Roma

Slide P.1

Progettazione del software
Design di applicazioni in Python

Implementazione del diagramma UML delle classi ristrutturato

Vincoli di integrità

Vincoli globali

Implementazione dei vincoli di integrità: vincoli globali

Un vincoli globale coinvolge oggetti e/o link di classi e/o associazioni diverse

La sua implementazione richiede un'attività di progettazione *ad hoc* secondo i passi seguenti:

1. **Comprendere se il vincolo è soddisfatto all'avvio del sistema, quando nessun oggetto/link esiste**

   —> Se no, bisogna fare in modo che alcuni oggetti e/o link necessari al soddisfacimento del vincolo siano creati automaticamente all'avvio

2. **Isolare quali sono i metodi che possono portare il sistema da uno stato che soddisfa il vincolo ad uno che lo viola**, ad es., quelli per la creazione di oggetti e link, i metodi set() delle varie class, i metodi che implementano le operazioni di use-case

   —> Equipaggiare il codice di tali metodi affinché si assicurino che il vincolo resti soddisfatto. In caso contrario, tali metodi dovranno sollevare opportune eccezioni

Implementazione dei vincoli di integrità: vincoli globali

- Ogni ricovero r nasce con r.periodo illimitato a destra e, una volta terminato, il suo periodo non può essere modificato
- Non devono esistere due ricoveri per lo stesso letto che si sovrappongono nel tempo
- Associazione ric_let a resp. doppia, con gestione asimmetrica da Ricovero

```
class Ricovero:
    _periodo:TimeRange # nasce illimitato a destra
    _letto:Letto
    def periodo(self)->Timerange:
        return self._periodo
    def letto(self)->Letto:
        return self._letto
    def set_end(self, end:strIdatetime)->None:
        if self.periodo().end():
            raise RuntimeError("Ricovero già terminato")
            # Se fosse possibile modificare l'istante di inizio o
            # di fine di un ricovero già terminato, dovremmo
            # controllare il vincolo anche qui!
        self._periodo = TimeRange(self._periodo.start(), end)
    def set_letto(self, letto:Letto)->None:
        if not letto:
            raise ValueError("letto cannot be None")
        if self._letto:
            # Rimuovi il vecchio link
            self._letto._remove_ricovero(self)
        self._letto = letto
        self._letto._add_ricovero(self)
    def __init__(self, start:strIdatetime, letto:Letto):
        if not start:
            raise ValueError("start cannot be None")
        self._periodo = TimeRange(start=start, end=None)
        self._letto = None
        self.set_letto(letto)
```

```
class Letto:
    _codice:str
    _ricoveri:set[Ricovero]
    def codice(self)->str:
        return self._codice
    def ricoveri(self)->frozenset[Ricovero]:
        return frozenset(self._ricoveri)
    def __init__(self, codice:str):
        if not codice: raise ValueError("codice cannot be None")
        self._codice = codice
        self._ricoveri = set()
    def _add_ricovero(self, ricovero:Ricovero)->None:
        if not ricovero: raise ValueError("ricovero cannot be None")
        # Esercizio: usare i generatori per evitare di generare tutta la lista
        if any([True for r in self.ricoveri() if r.periodo().intersects(ricovero.periodo())]):
            raise ValueError("ricovero si sovrappone con altro ricovero per questo letto")
            # Esercizio: implementare un indice dei ricoveri r associati al letto self
            # secondo i valori di r.periodo()
        self._ricoveri.add(ricovero)
```

ITC INFORMATION AND COMMUNICATIONS TECHNOLOGY ACADEMY

MODULO: Progettazione
UNITÀ: Progettazione.1

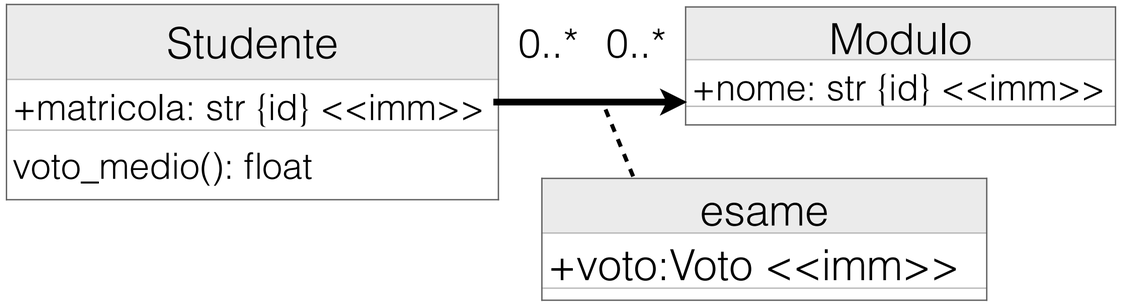
Prof. Toni Mancini
Dipartimento di Informatica
Sapienza Università di Roma

Slide P.1

Progettazione del software
Design di applicazioni in Python

Implementazione del diagramma UML delle classi ristrutturato

Operazioni di classe



Metodo nella class Python

```
import statistics

class Studente:
  …
  def voto_medio(self)->floatlNone:
    try:
      return statistics.mean(  [ l.voto() for l in self.esami() ] )
    except statistics.StatisticsError:
      return None
```

ITC INFORMATION AND COMMUNICATIONS TECHNOLOGY ACADEMY

MODULO: Progettazione
UNITÀ: Progettazione.1

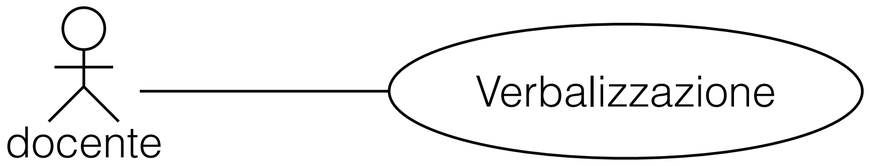
Prof. Toni Mancini
Dipartimento di Informatica
Sapienza Università di Roma

Slide P.1

Design di applicazioni in Python

Implementazione delle specifiche ristrutturate degli use-case

Implementazione delle operazioni di use-case



Un modulo Python per ogni use-case, con una funzione per ogni operazione

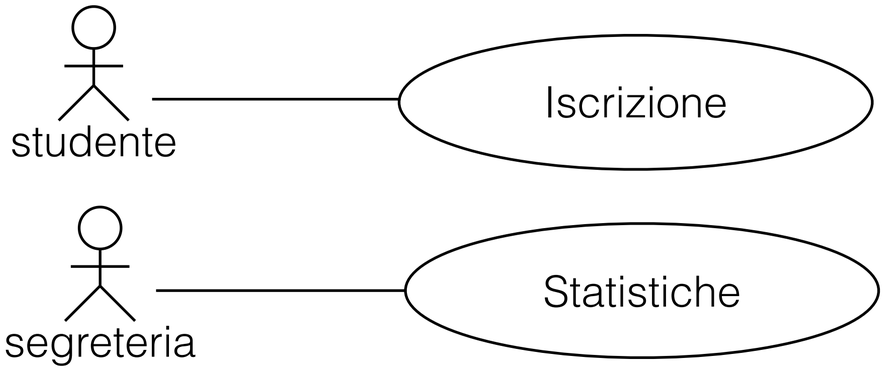
- semplici funzioni, non metodi di classe
- visibilità: public per le funzioni che implementano le operazioni di use-case; private per le funzioni di supporto
- **l'utente non deve mai avere accesso** ad oggetti del sistema che potrebbe modificare senza controllo
  - **accorgimenti**, ad es.:
    - **chiediamo** come argomenti **valori di attributi {id}** e non oggetti
    - …

- Studente.matr è {id} per la classe
- Corso.nome è {id} è la classe

verbalizzazione.py

```
def verbalizzazione(matr:str, mod:str, v:Voto)->None:
    s = Studente.get(matr) # matricola è {id} per la classe Studente
    if not s: raise ValueError(f"studente di matricola {matr} non trovato")
    m = Modulo.get(mod) # nome è {id} per la classe Modulo
    if not m: raise ValueError(f"modulo di nome {nome} non trovato")
    s.add_link_esame(m, v)
```

Implementazione delle operazioni di use-case (cont.)



Un modulo Python per use-case, con una funzione per ogni operazione

- …
- **l'utente non deve mai avere accesso** ad oggetti del sistema che potrebbe modificare senza controllo
  - **accorgimenti**, ad es.:
    - chiediamo come argomenti valori di attributi {id} e non oggetti
    - **restituiamo viste immutabili** di oggetti.

```
class Studente:
  …
  def view(self)->Studente.View:
    return Studente.View(self)
  # vista in sola lettura di oggetti della classe, mediante
  incapsulamento
  class View:
    __studente:Studente
    def __init__(self, s:Studente):
      self.__studente = s
    def matricola(self)->str:
      return self.__studente.matricola()
    def esami(self)->frozenset[esame._link]:
      return self.__studente.esami()
    def media(self)->float:
      return self.__studente.media()
    # Accesso non consentito ai metodi setter di Studente
```

iscrizione.py

```
def iscrizione(nome:str, cognome:str, nascita:datetime.date,
c:Citta)->Studente.View:
  return Studente(nome, cognome, nascita, c).view()
```

statistiche.py

```
def media(s:Studente)->None:
  return s.media()
def promossi(m:Modulo.View)->set[Studente.View]:
  return set( I().studente().view() for I in m.esami() )
```