DIETI UNINA UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II
DIPARTIMENTO DI INGEGNERIA ELETTRICA E DELLE TECNOLOGIE DELL'INFORMAZIONE
Corso di Laurea in Ingegneria Informatica

Corso di Ingegneria del Software

Prof. Roberto Pietrantuono

Il linguaggio UML

Diagramma delle classi

Sommario

- Modellazione delle classi in analisi, progettazione e implementazione
- Diagramma delle classi UML
- Esempi

Diagramma delle classi (CD)

Il **diagramma delle classi UML (*Class Diagram*, CD)** mostra le classi di oggetti e le relazioni tra esse.

Si tratta di una vista **statica.**

Può essere prodotto:

- in fase di analisi e specifica dei requisiti, mostrando classi di oggetti del dominio del problema;
- in fase di progettazione di alto livello, mostrando classi di oggetti del dominio della soluzione;
- in fase di progettazione di dettaglio o ("UML in fase di implementazione").

Gli elementi del CD corrispondono a concetti fondamentali del paradigma *object-oriented* (classi, oggetti, ereditarietà, etc.)

Diagramma delle classi – fase di analisi

In fase di analisi, un diagramma delle classi è una vista concettuale delle entità nel dominio del problema.

Cattura i concetti principali del dominio, a prescindere:

- da come essi saranno rappresentati in archivi o comunque in un sistema di elaborazione,
- dalle funzionalità che opereranno su di essi (→software).

*L'attenzione dovrà essere rivolta alla specifica di **"quali"** entità il sistema software tratta e delle relazioni tra di esse, e non **"come"** il software dovrà archiviarle o elaborarle.*

Diagramma delle classi – fasi di progetto e implementazione

In fase di progetto, un diagramma delle classi descrive le entità del sistema e le relazioni tra esse (ancora prescindendo dalla implementazione); è una delle viste statiche del sistema progettato.

In fase di implementazione: le classi e le relazioni indicano la reale struttura del software.

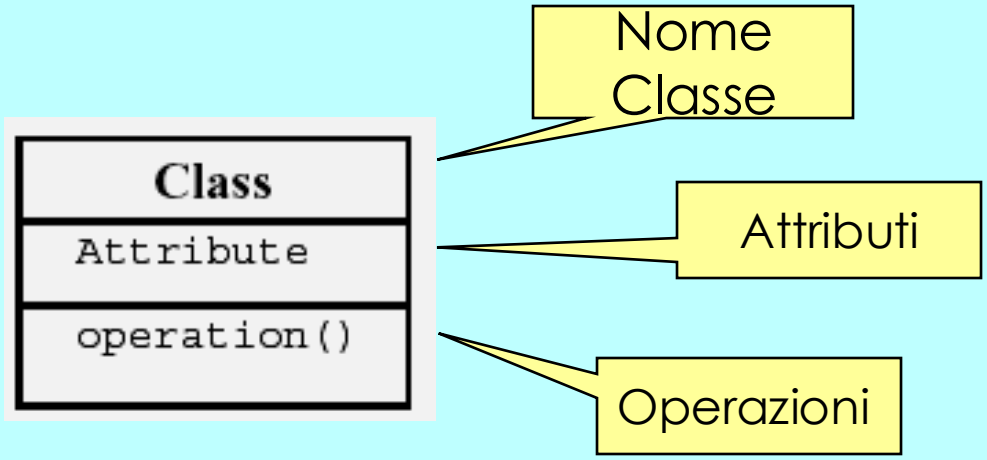
Nei CD di progettazione e implementazione trovano posto anche classi e relazioni che riguardano la realizzazione del sistema software, e non hanno diretta corrispondenza con classi e relazioni del CD di analisi.

Sintassi UML per le classi

Sintassi per le classi nei diagrammi UML

Un CD mostra gli **elementi che esistono, le loro proprietà/attributi e responsabilità/operazioni, e le relazioni** con le altre entità (non mostra informazioni temporali).

Classe: è il descrittore di un *insieme* di oggetti con le stesse proprietà (attributi), comportamento (operazioni) e relazioni con altri oggetti.



Nella rappresentazione UML il nome della classe deve essere indicato; gli attributi e le operazioni sono opzionali

Classe e Oggetti

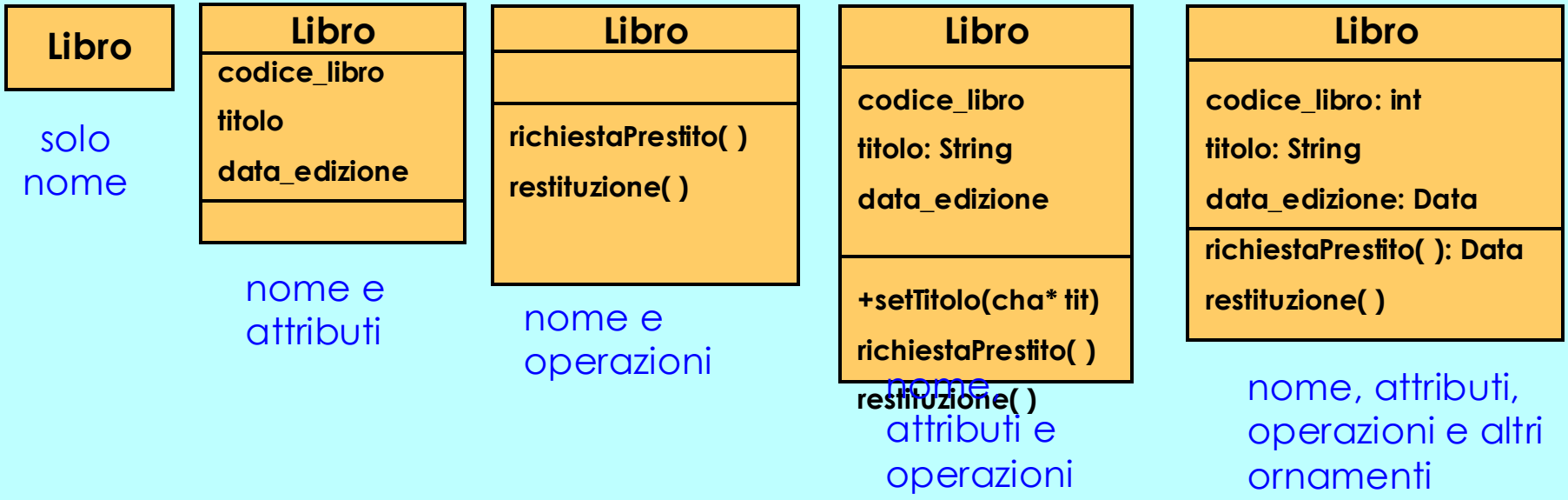
- Una classe è il **descrittore di un insieme di oggetti** che condividono attributi, operazioni, relazioni.
- Ogni oggetto è una **singola istanza** di una classe
  - Ha specifici valori per gli attributi della classe, ossia
  - Ha uno **stato** (valore dei suoi attributi)
    - In un dato istante, due oggetti della stessa classe hanno in generale stati diversi
  - In base al proprio stato, due oggetti possono rispondere diversamente alla stessa operazione
    - Es. se si tenta di prelevare 100 Euro da un oggetto conto corrente, il risultato sarà diverso, secondo la disponibilità

Notazione UML

Una classe è rappresentata da un rettangolo con tre diverse sottosezioni: nome, attributi, operazioni.

Il nome è solitamente scritto in "UpperCamelCase" (inizia con una maiuscola e, se costituito da più parole, non sono previsti spazi, ma ogni parola inizia con la maiuscola).

Il nome è obbligatorio, le altre sottosezioni possono essere omesse.



Attributi

Gli attributi sono scritti in «lowerCamelCase»

Es.: *indirizzoResidenza*

Possono presentare vari ornamenti (visibilità, molteplicità, tipo, etc.)

Struttura generale:

visibilità **nome** molteplicità: tipo = valoreIniziale {proprietà}

opzionale

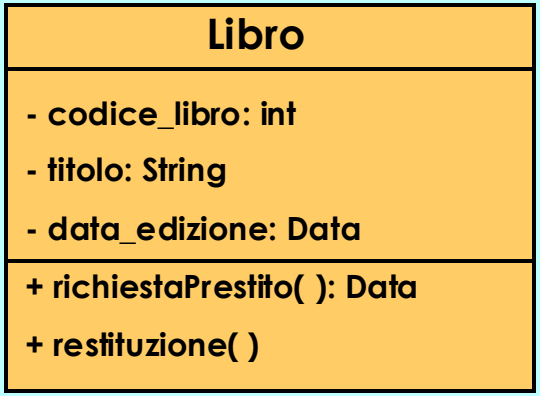
obbligatorio

opzionale

Ornamenti degli attributi: visibilità

visibilità nome molteplicità: tipo = valoreIniziale {proprietà}

- Sono ammessi tre livelli di visibilità degli attributi:
  - \+ **Livello pubblico**: L'utilizzo è esteso a tutte le classi
  - \# **Livello protetto**: L'utilizzo è consentito soltanto alle classi che derivano dalla classe originale
  - \- **Livello privato**: Soltanto la classe originale può utilizzare gli attributi e le operazioni definite come tali.



Ornamenti degli attributi: tipo

- Il **nome** dell'attributo é necessario
- Il **tipo** dell'attributo può essere un tipo primitivo (**int**, **double**, **char**, etc…), oppure il **nome di una classe** definita nello stesso diagramma

Ornamenti degli attributi: valore iniz.

visibilità nome : tipo [molteplicità] = valoreIniziale {proprietà}

valoreIniziale è il valore assegnato all'attributo per *default*

{**proprietà**} indica caratteristiche aggiuntive dell'attributo (ad esempio la sola lettura)

Il tipo può essere seguito dalla **molteplicità**, che indica indica il quantitativo degli attributi (ad esempio la dimensione per un array), ad esempio «Indirizzo: String[3]»

Alcuni valori possibili sono:

1 (uno e uno solo). È il valore di default.

0..1 (al più uno)

* (un numero imprecisato, eventualmente anche nessuno; equivalente a 0..*)

1..* (almeno uno)

Esempio: name: String [1] = "Untitled" {readOnly}

Operazioni

Sono le operazioni invocabili **sugli** oggetti istanza della classe.

La sintassi completa per la firma (*signature*) di una operazione è:

visibilità nome (lista parametri) : tipo-restituito {proprietà}

La **visibilità** e il **nome** seguono regole analoghe a quelle per gli attributi.

Lista parametri contiene nome e tipo dei parametri della funzione, secondo la forma:

direzione nome parametro: tipo = valore-di-default

direzione: input (***in***), output (***out***) o entrambi (***inout***). Il valore di *default* é *in.*
nome, tipo e valore di default sono analoghi a quelli degli attributi
Tipo-restituito é il tipo del valore di ritorno

Esempio:

+ saldoAllaData(in giorno: Data): float

Relazioni tra classi nel diagramma UML delle classi

Relazioni tra classi

- Le classi (o gli oggetti) in un diagramma delle classi (degli oggetti) non sono in generale isolate, ma in relazione con altre entità (oggetti) con legami di varia natura, detti "**relazioni**".
- Le principali relazioni tra le classi sono:
  - Relazione di **generalizzazione-specializzazione**
  - Relazione di **contenimento (lasco/stretto)**
  - Relazione di **associazione**
- Si possono inoltre esprimere relazioni di **dipendenza**

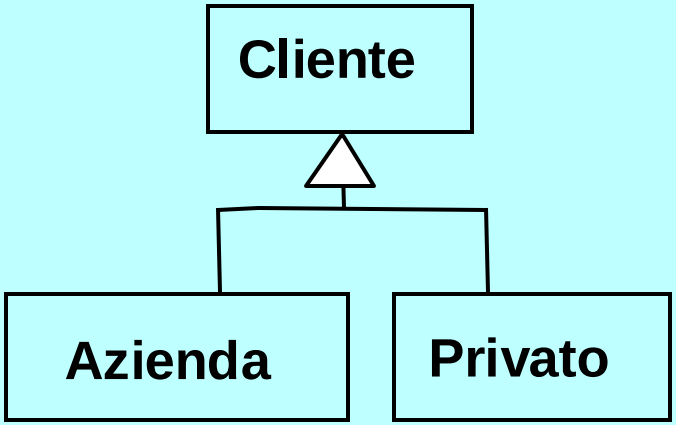
Relazione Gen-Spec

Generalizzazione-specializzazione

La relazione di **generalizzazione-specializzazione (o gen-spec)**, tipica dell'orientamento agli oggetti (*object-orientation*), corrisponde al legame tipo-sottotipo.
Essa **indica legami tra classi del genere "è un" (is-a)**:

Es.: un ContoDiDeposito *è un* ContoBancario

La **superclasse** raccoglie caratteristiche e comportamenti comuni agli elementi delle **sottoclassi**.



L'azienda è ("is a") un cliente

I clienti sono sia aziende sia privati

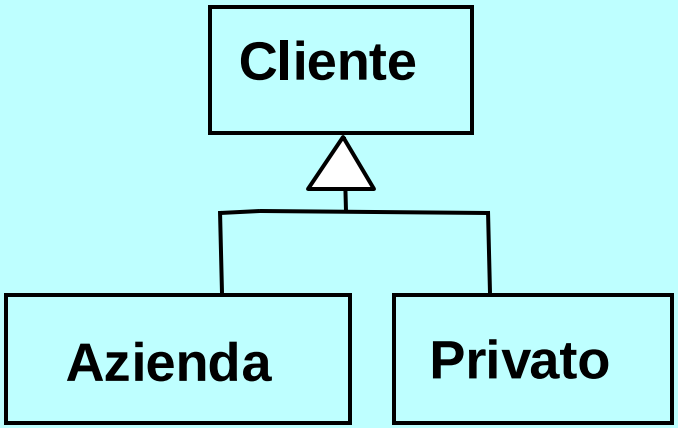
Generalizzazione-specializzazione

Le relazioni gen-spec descrivono **gerarchie di ereditarietà**

La *classe derivata (sottoclasse)* eredita le caratteristiche della *classe base (superclasse).*
Gli attributi e le operazioni della classe base apparterranno anche alla classe derivata.
La classe derivata può:
- aggiungere altri attributi e operazioni
- specializzare operazioni della classe base

Principio di Sostituibilità

Principio di Sostituibilità: Ogni elemento della classe Azienda (o Privato) è anche un elemento della classe Cliente.
Ne consegue che il software potrà riferirsi al tipo Cliente, senza dover distinguere se esso è un'Azienda o un Privato.



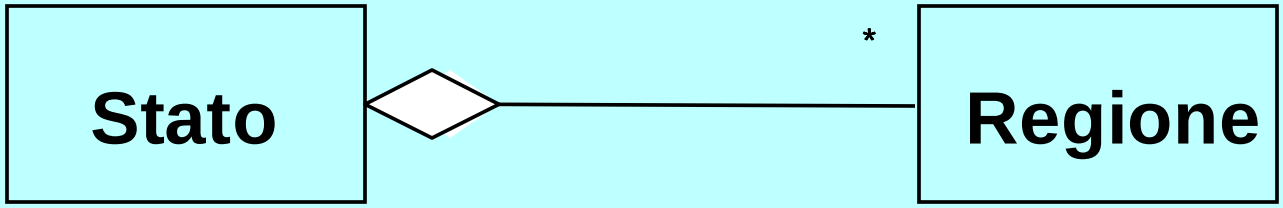
L'azienda è ("is a") un cliente
Un privato è un cliente

I clienti sono aziende o privati

Relazione di contenimento (aggregazione, composizione)

Relazione di contenimento

Un tipo di legami tra classi è la relazione di **contenimento**, che rappresenta il legame tra un insieme (il "tutto") e le sue parti (***legame tutto-parti***), ovvero tra un contenitore e il contenuto (***legame contenitore-contenuto***)



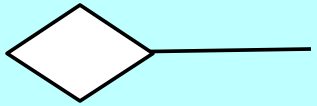
Il legame è modellato in UML con un segmento che unisce due classi, con un rombo dal lato del contenitore. Il rombo è vuoto o pieno, secondo il tipo di contenimento

Aggregazione e composizione

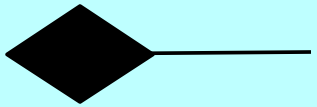
Il contenimento può infatti essere

- ***debole*** (o lasco), detto anche ***aggregazione***: si ha quando la parte mantiene la sua identità quando entra a far parte del tutto (Es.: auto – motore)
  - La vita della parte è indipendente da quella del tutto
- ***stretto,*** detto anche ***composizione***: se la parte perde la sua identità quando entra a far parte del tutto (Es.: pane – farina)
  - La vita della parte è dipende da quella del tutto

L'**aggregazione** è modellata da un **rombo vuoto**.



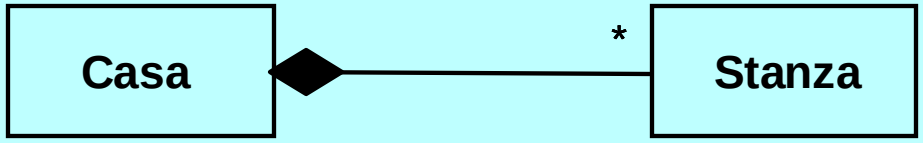
La **composizione** è modellata da un **rombo pieno**.



Ciclo di vita degli oggetti in una composizione

- La relazione di **composizione** indica che l'oggetto **contenuto** non ha una vita propria ma esiste in quanto parte dell'oggetto **contenitore**.
- Se l'aggregato viene distrutto, anche le sue parti saranno distrutte (le parti non esistono senza il tutto)
- L'oggetto **contenitore** è responsabile della costruzione e distruzione dell'oggetto **contenuto**

- Es.: Un palazzo contiene delle stanze.
- Il modello UML di figura specifica che non ha senso parlare di stanze se non in quanto facenti parte di una casa



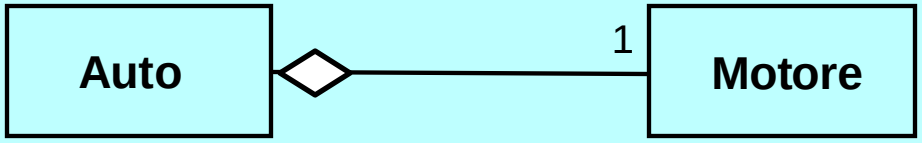
Semantica della composizione

- Ogni parte può **appartenere ad un solo composito** per volta
- Se il composito viene distrutto, deve **distruggere tutte le sue parti** o cederne la responsabiltà a qualche altro oggetto
- Il composito può rilasciare una sua parte, a patto che un altro oggetto si prenda la relativa responsabilità

Ciclo di vita degli oggetti in una aggregazione

- La relazione di **aggregazione** indica l'indipendenza del ciclo di vita dell'oggetto **contenuto** da quello dell'oggetto **contenitore**.
- L'oggetto **contenuto** può quindi esistere anche indipendentemente dal **contenitore**.
- L'oggetto **contenitore** NON è necessariamente responsabile della costruzione e distruzione dell'oggetto **contenuto**

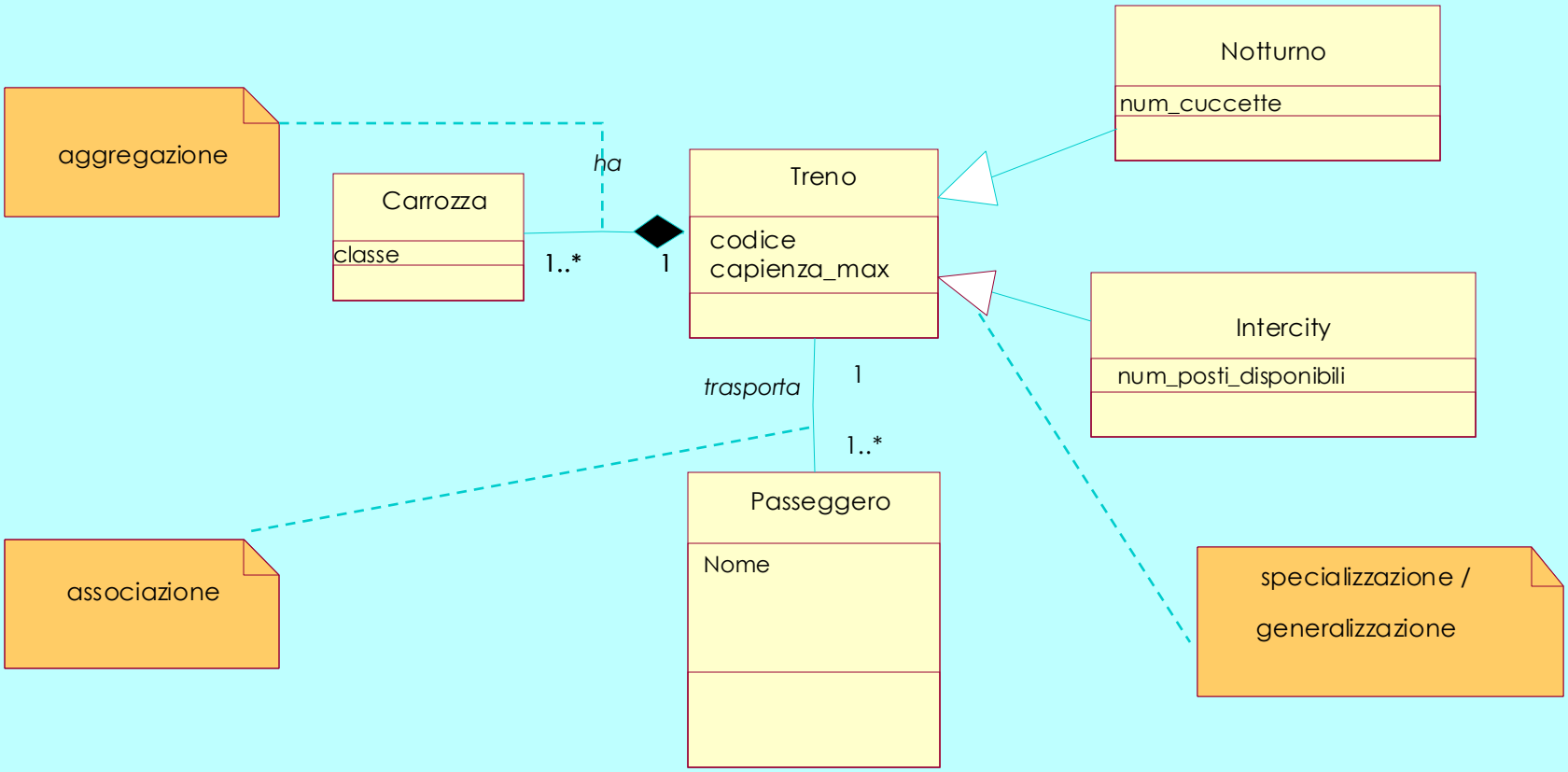
- Es.: Un'auto ha un motore.
- Il modello UML di figura specifica che il motore di un'auto può vivere anche indipendentemente dall'auto (per es., si può rottamare l'auto ma prima smontarne il motore e montarlo su un'altra auto)



Semantica dell'aggregazione

- Le parti possono **esistere indipendentemente** dall'aggregato
- In alcuni casi l'aggregato può esistere indipendentemente dalle parti, in altri casi no
- L'aggregato è in qualche modo incompleto se mancano alcune delle sue parti
- È possibile che più aggregati **condividano** una stessa parte
- L'aggregazione è transitiva
- L'aggregazione è asimmetrica

Un esempio completo



Relazioni tra classi

Associazioni

Relazione di associazione

L'associazione tra due (o più) classi esprime un **legame semantico tra le istanze (oggetti) delle classi**.

Tutti i legami tra (istanze di) classi che non sono gen-spec né di contenimento sono associazioni

Esempio: **Ogni Automobile ha per proprietario una Persona**

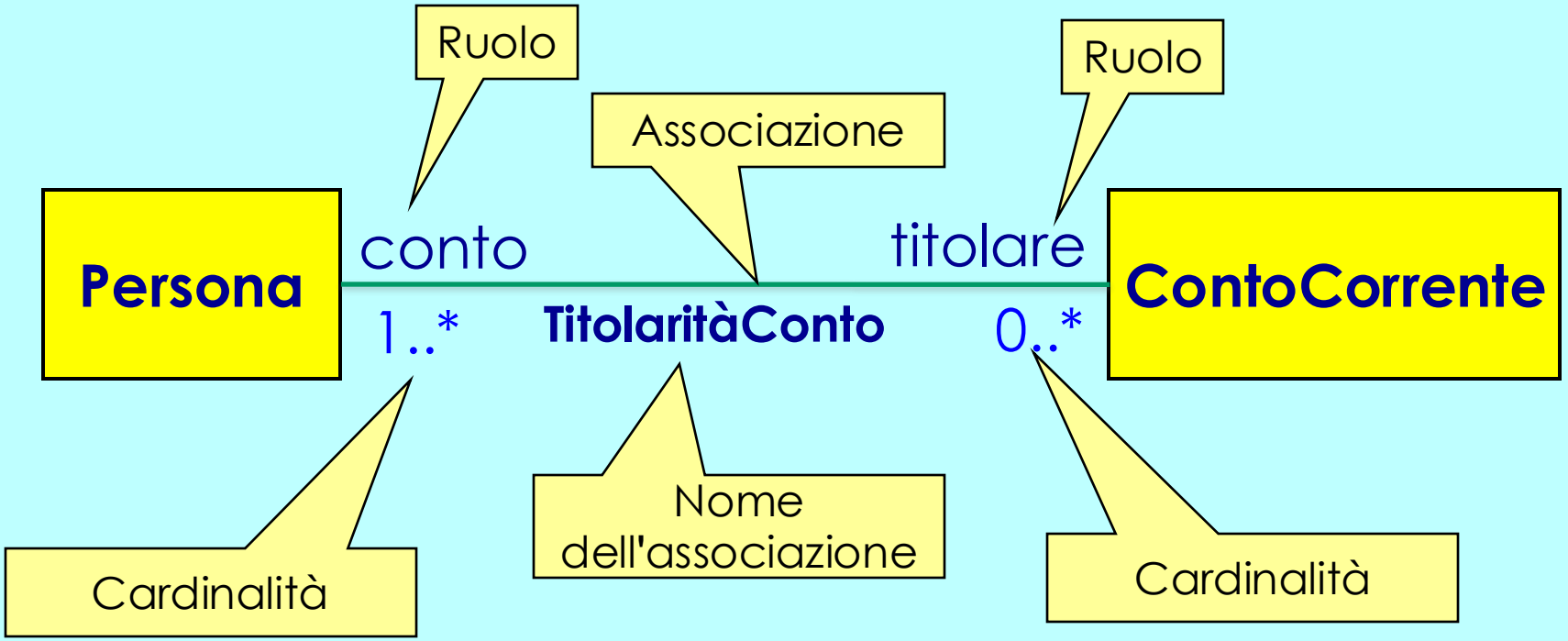
Una associazione è caratterizzata da:

- ✓ un **nome**: esprime il legame semantico tra le classi associate
- ✓ un eventuale **ruolo** giocato da ciascuna delle parti associate
- ✓ la **molteplicità** dell'associazione
  - ✓ esprime la cardinalità delle connessioni tra oggetti delle classi associate, che può essere di vari tipi

- uno-a-uno **- uno-a-molti** **- molti-a-molti**
- zero-a-molti **- …**

Sintassi UML per l'associazione

- Tutti i legami tra (istanze di) classi che non sono gen-spec né di contenimento sono associazioni



Molteplicità

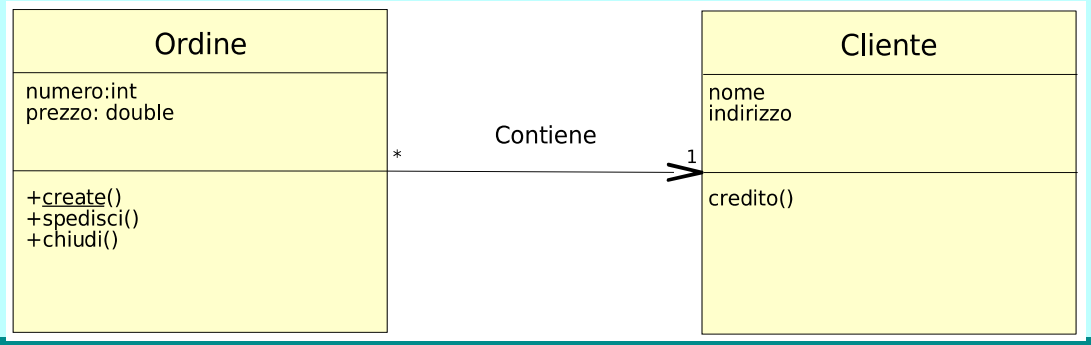
Sono della forma:

- 0,1 Zero (partecipazione opzionale) o uno
- 1 Esattamente 1
- 0..* Zero o più
- * Zero o più
- 1..* Uno (partecipazione obbligatoria) o più
- 1..6 Da 1 a 6
- 1..4,7 Da 1 a 4, oppure 7

Navigabilità dell'associazione

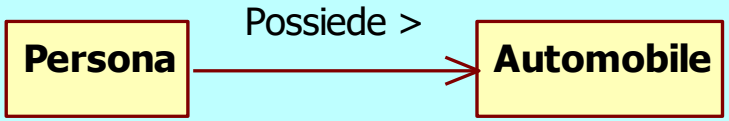
- Da un punto di vista concettuale, una associazione non ha un verso di percorrenza (una direzionalità)
  - Se A è legato a B, B è legato ad A

- Nonostante ciò, UML offre la possibilità di specificare una direzionalità
  - Il segmento diventa una freccia uni- o bi-direzionale
- Se specificata, la direzionalità attribuisce alla classe origine del verso di percorrenza la responsabilità di tenere traccia dell'associazione (a entrambe le classi in caso di freccia bidirezionale)



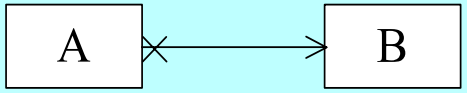
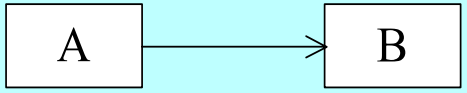
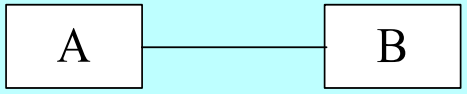
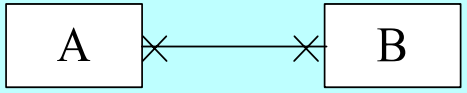
Verso di navigazione

- Il verso di navigazione di una associazione è un'informazione utile soprattutto in fase di **progetto**
- Indica in quale direzione è possibile reperire le informazioni



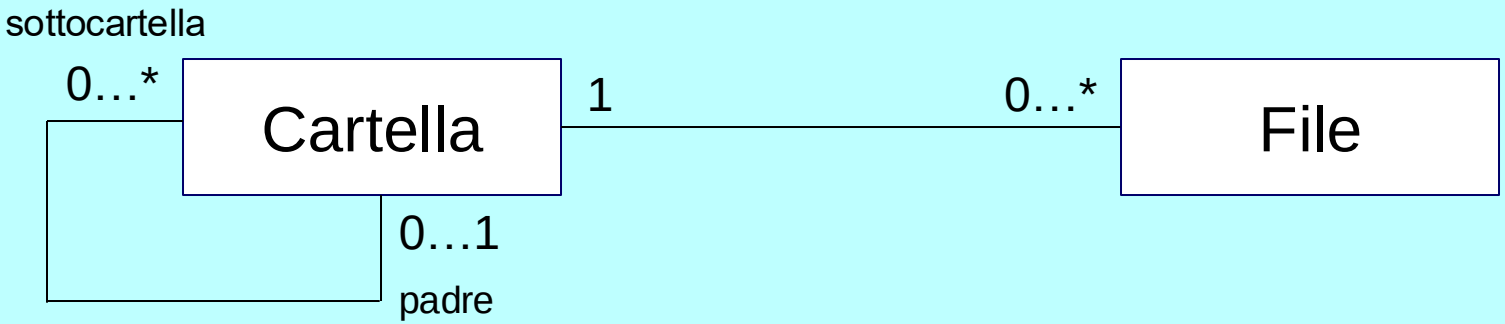
- L'esempio specifica che, nota una persona, si vuole poter risalire direttamente alle automobili che possiede
- Viceversa, data una automobile non si vuole poter risalire direttamente al suo proprietarios
- Di solito, il verso di navigazione rappresenta una scelta di progetto, per cui non è presente nei diagrammi concettuali (di analisi)

Idiomi di navigabilita suggeriti da UML 2.0

Sintassi UML 2	Idioma 1: Navigabilità UML 2.0 stretta	Idioma 2: Nessuna navigabilità	Idioma 3: standard nella pratica
	Da A a B è navigabile Da B a A è navigabile		
	Da A a B è navigabile Da B a A non è navigabile		
	Da A a B è navigabile Da B a A è indefinito		Da A a B è navigabile Da B a A non è navigabile
	Da A a B è indefinito Da B a A è indefinito	Da A a B è indefinito Da B a A è indefinito	Da A a B è navigabile Da B a A è navigabile
	Da A a B non è navigabile Da B a A non è navigabile		

Associazione riflessiva

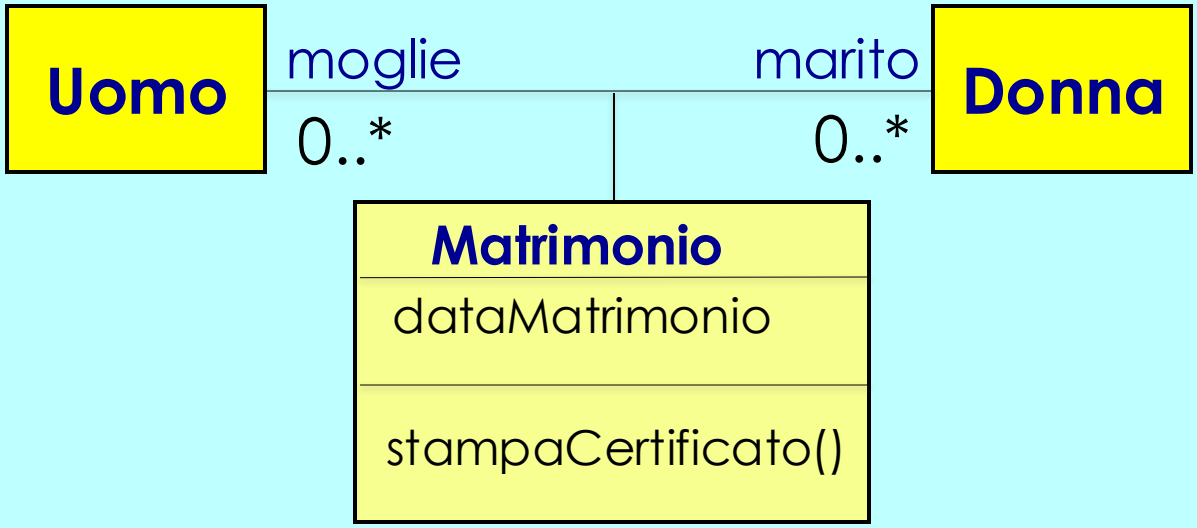
- L'associazione può essere riflessiva
  - Oggetti della classe possono avere collegamenti con oggetti della stessa classe



- Le relazioni di associazione si traducono in C++ con una o più variabili membro di tipo puntatore alla classe associata (a seconda della molteplicità dell'associazione), che consente la navigabilità

Classi associative

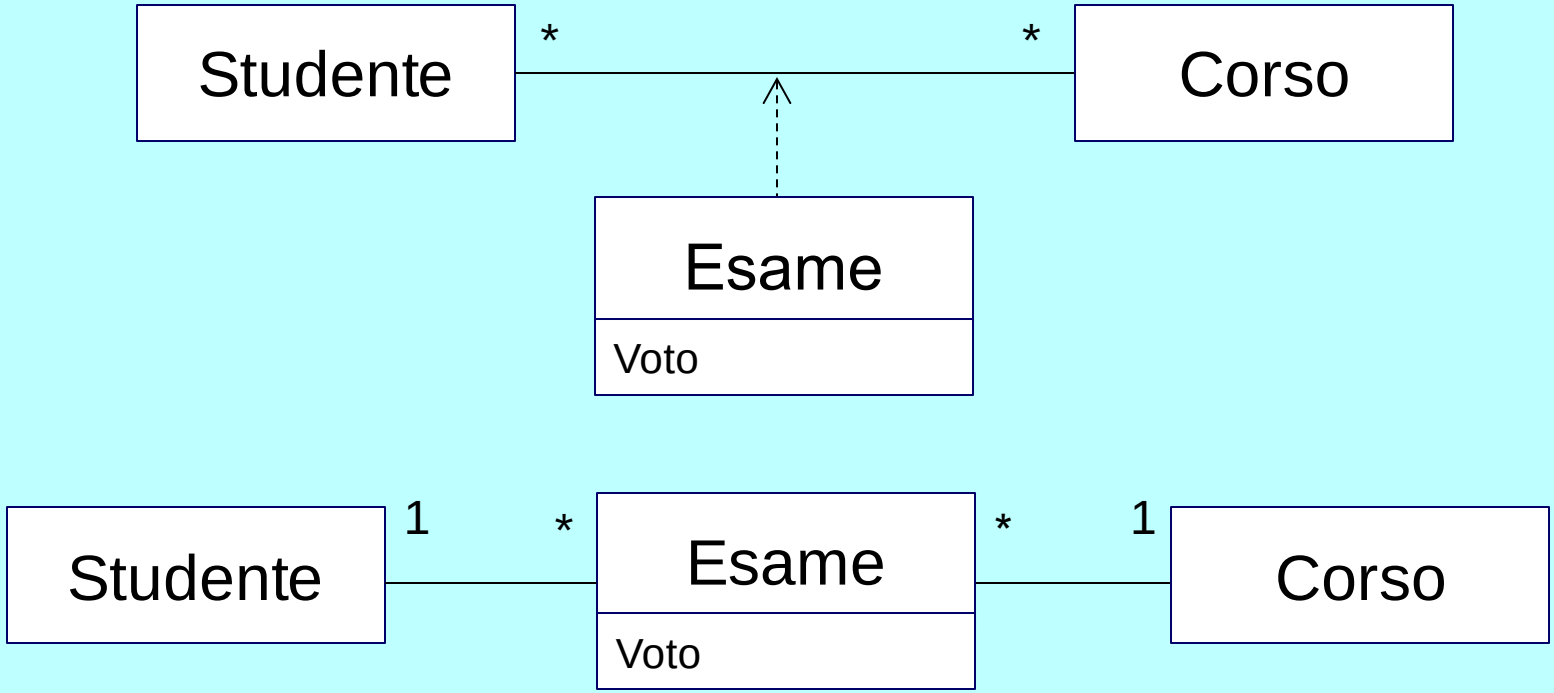
- Nelle associazioni molti-a-molti, possono esserci degli attributi **non assegnabili a nessuna delle due classi** della relazione



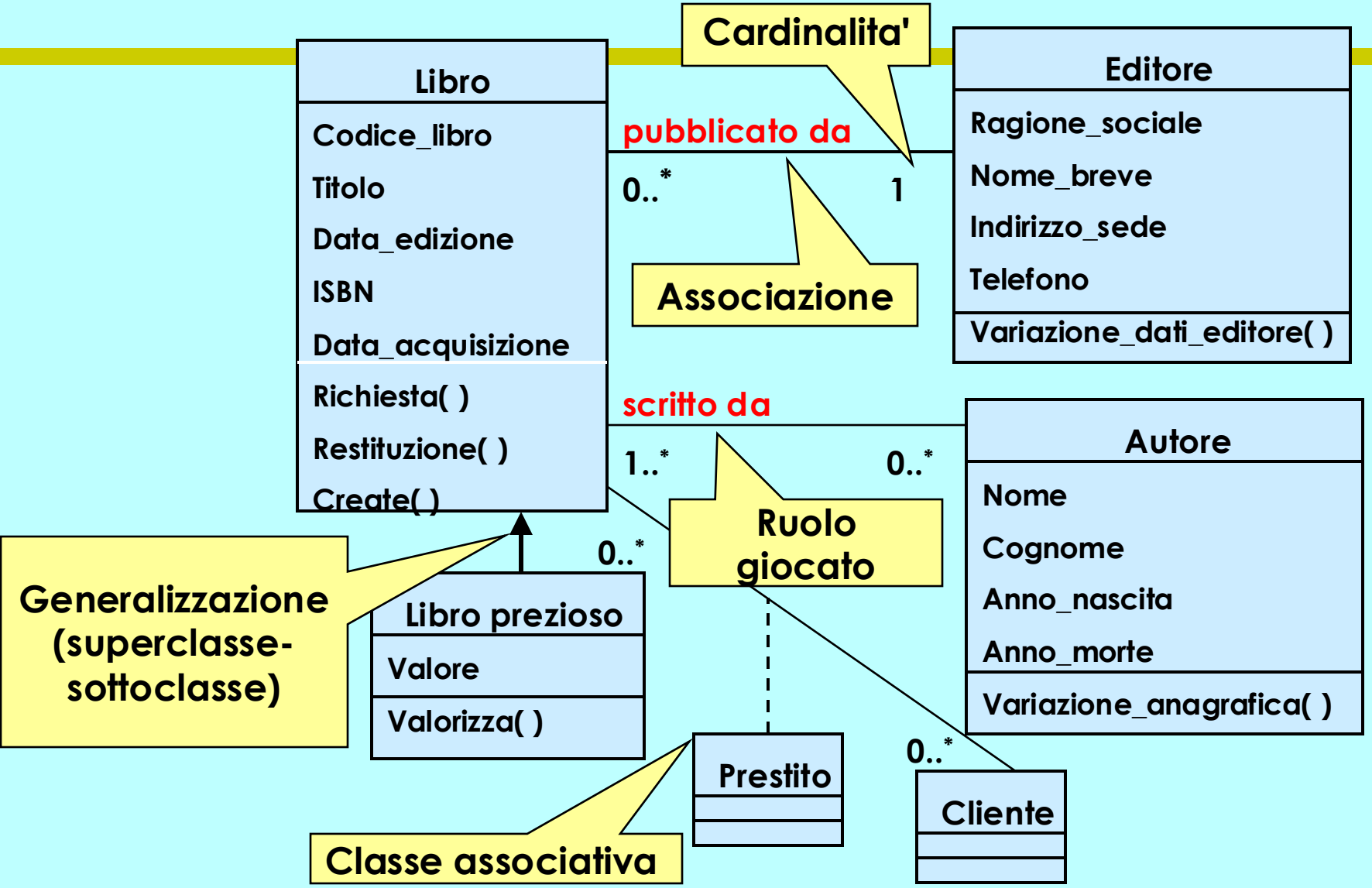
- La **classe associativa** connette due classi e definisce un insieme di **caratteristiche proprie della associazione stessa**
- Le istanze dell'associazione sono **collegamenti tra oggetti**, dotati di attributi e operazioni

Classe associativa

- Per **reificare** (rendere reale) una classe associativa sarà opportuno (tipicamente, in fase di progettazione) trasformarla in una vera classe



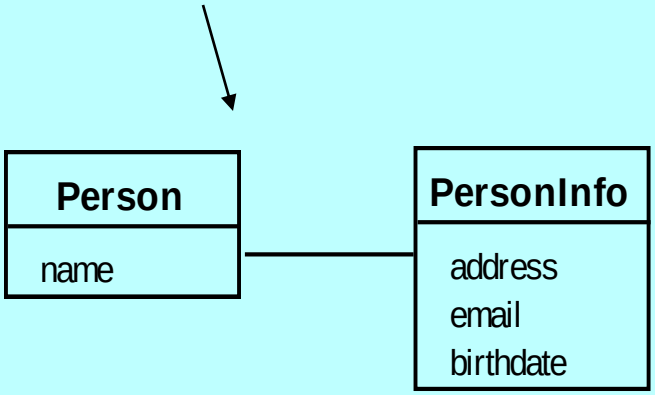
Esempio


Cardinalita'
Libro
Codice_libro
Titolo
Data_edizione
ISBN
Data_acquisizione
Richiesta( )
Restituzione( )
Create( )
pubblicato da
0..*
1
Associazione
Editore
Ragione_sociale
Nome_breve
Indirizzo_sede
Telefono
Variazione_dati_editore( )
scritto da
1..*
0..*
Ruolo giocato
Autore
Nome
Cognome
Anno_nascita
Anno_morte
Variazione_anagrafica( )
Generalizzazione (superclasse-sottoclasse)
0..*
Libro prezioso
Valore
Valorizza( )
Prestito
0..*
Cliente
Classe associativa

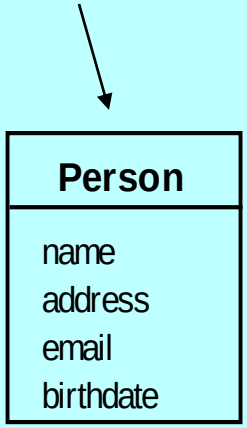
Contro-esempio

- A volte le associazioni *uno a uno* sono inutili

Evitare …



migliore soluzione!



Relazioni tra classi

Dipendenze

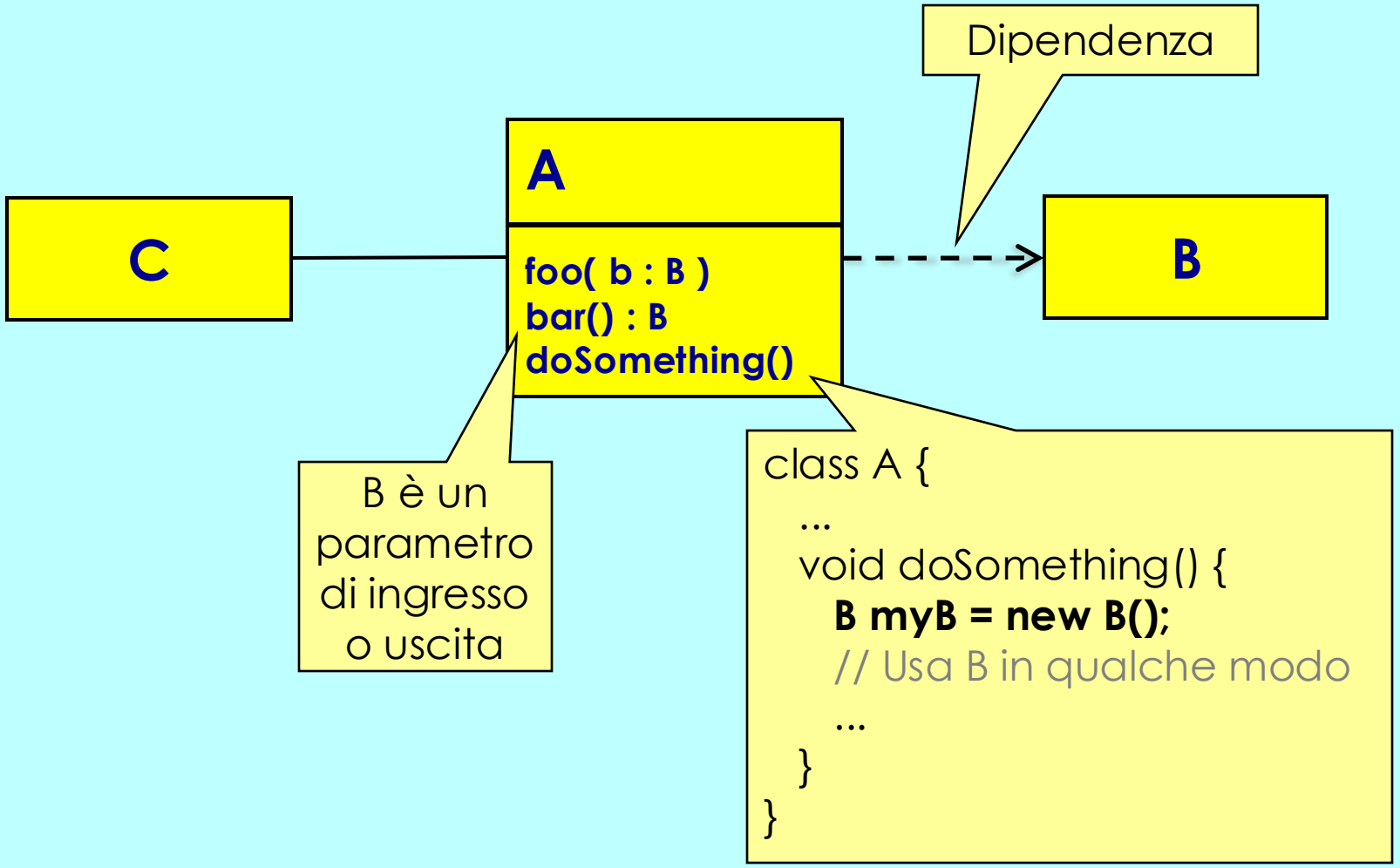
Relazione di dipendenza

La relazione di **dipendenza** tra due classi esiste laddove la **modifica** di una classe **impatta sull'altra** o le fornisce informazioni, ma **non ricade in un gen-spec, in una associazione o in un contenimento**.
Si rappresenta con una freccia tratteggiata, appositamente stereotipata

Esempi:
- Un oggetto della classe A viene usato come parametro di ingresso verso una operazione della classe B
- La classe A restituisce un oggetto di classe B
- Una operazione della classe A usa dentro la sua implementazione un oggetto di classe B, ma non ha attributi di classe B

La relazione di dipendenza è anche utilizza tra package e in altri casi avanzati

Sintassi UML per la dipendenza



Individuare
le classi di analisi

Indicazioni – Class Diagram di analisi

- Usare sempre il linguaggio del «dominio di business»
- Ogni diagramma deve essere **utile a illustrare** un qualche comportamento desiderato del sistema («tell a story»)
- **Evitare i dettagli** sul funzionamento e della soluzione (da rimandare alla fase di progetto)
- Minimizzare l'**accoppiamento**
- Usare al minimo l'**ereditarietà** (se esiste una gerarchia naturale da modellare); i problemi di riuso vanno rimandati
- Il modello è **utile per gli stakeholders**?

Analisi dei nomi e dei verbi

- Una tecnica diffusa per individuare le classi in fase di analisi dei requisiti consiste nell'**analisi dei nomi e dei verbi**
- Essa si basa sull'esame della descrizione testuale del dominio del problema
  - Analisi dei requisiti espressi in forma testuale
- Analizzare sinonimi e omonimi
- Disambiguare i termini

Individuare le classi di analisi

- Nei requisiti testuali, i **sostantivi** indicano candidati per:
  - Classi
  - Attributi
  - Ruoli
  - Attori (→ diagramma dei casi d'uso)
- I **verbi** indicano candidati per:
  - Relazioni tra classi (gen-spec, contenimento, associazione)
  - Responsabilità delle classi
  - Funzionalità del software (→ diagramma dei casi d'uso)

Individuare le classi di analisi

- I sostantivi indicano candidati classi, attributi e ruoli
  - Il sostantivo indica un **insieme di elementi** che sono esemplari (cioè individui distinti)? → classe
    - Es: ContoCorrente
  - Il sostantivo indica una **proprietà di un elemento** di un insieme? → attributo
    - Es.: Saldo (proprietà di un ContoCorrente)
  - Il sostantivo indica il nome che una istanza di una classe assume quando la si considera come **parte di un legame** con altri oggetti? → ruolo
    - Es.: La *Persona* che ha aperto un *ContoCorrente* ne diventa il *Titolare*

Individuare le relazioni tra classi

- Le espressioni verbali che coinvolgono più classi indicano possibili relazioni tra esse
- Cercare nell'ordine:
  - Espressioni del tipo "è un" → Gen-Spec
  - Espressioni del tipo "è fatto di", "comprende", … → Contenimento
  - Ogni altra espressione → Associazione
- N.B.: i requisiti che descrivono possibili relazioni tra classi rappresentano fatti della realtà che si sta modellando
  - Es.: è un fatto della realtà di interesse che quella persona sia titolare di quel conto corrente

Individuare le classi di analisi

- I verbi indicano anche possibili operazioni delle classi
  - Attenzione: le operazioni di una classe sono tutte e sole quelle che si possono invocare sugli oggetti della classe
    - Non quelle fatte dagli oggetti della classe
  - Es: L'utente effettua il login – **login NON è una responsabilità della classe Utente**

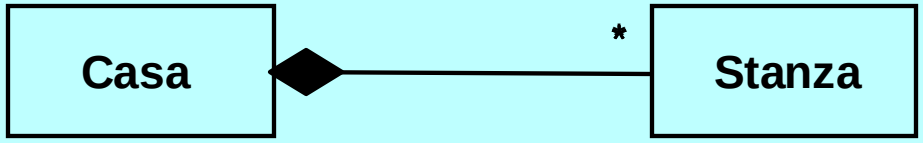
- Attenzione: Le frasi che esprimono funzionalità del software specificano casi d'uso – NON vanno modellate nel diagramma delle classi di analisi

Traduzione delle relazioni tra classi (C++, Java)

Traduzione della composizione in un linguaggio ad oggetti

- La composizione si traduce in un linguaggio ad oggetti:
  - Aggiungendo alla classe contenitore una variabile membro del tipo della classe contenuto
  - Implementando un costruttore del contenitore in modo da richiamare il costruttore del contenuto

- L'utente della classe contenitore ha la responsabilità di instanziarne un oggetto, richiamando il suo costruttore con tutti i valori necessari all'inizializzazione sia del contenitore sia del suo contenuto.



Codifica della composizione in C++ e in Java

In particolare, la **composizione** si può realizzare in C++ o in Java come segue:

- in **C++**: il contenitore C ha come variabile membro privata un **oggetto** del tipo contenuto X
- in **Java**: il contenitore C ha una variabile membro privata di tipo **riferimento** ad un oggetto del tipo contenuto X

In entrambi i casi il contenitore C dovrà avere un costruttore che costruisce un oggetto c di tipo C con dentro anche il suo contenuto x di tipo X.

Esempio di composizione (uno-a-uno) in C++

```
//file: Contenuto.h
class Contenuto {
  public:
     Contenuto(String s);

  private:
     String nomeContenuto;
};
```

```
//file: Contenuto.cpp
#include "Contenuto.h"
Contenuto::Contenuto(String s) : nomeContenuto(s){}
```

```
//file: Contenitore.h
#include "Contenuto.h”
class Contenitore {
 public:
      Contenitore(String, String);
      ~Contenitore();
 private:
   Contenuto c;     //N.B.: è un
    // oggetto, non un puntatore!
  String nomeContenitore;
};
```

```
//file: Contenitore.cpp
#include "Contenitore.h"

//Costruttore: chiama anche il costruttore del contenuto
Contenitore::Contenitore(String a, String x) :
                nomeContenitore(a),c(x){}

//Distruttore: chiama anche il distruttore del contenuto
Contenitore::~Contenitore() { delete &c;}
```

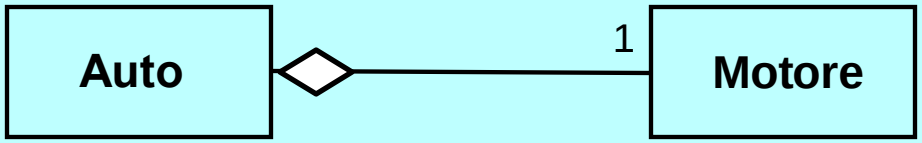
Esempio di composizione (uno-a-uno) in Java

```
class Contenitore {
   public Contenitore(String s1, String s2) {
      nomeContenitore = s1;
      c = new Contenuto(s2);
   }

   private String nomeContenitore;
   private Contenuto c;          // composizione: variabile membro di tipo Contenuto
};
```

```
class Contenuto {
    public Contenuto(String s) {nomeContenuto=s};

    private String nomeContenuto;
};
```

Traduzione dell'aggregazione in un linguaggio ad oggetti

- L'aggregazione si traduce in un linguaggio ad oggetti:
  - Inserendo nella classe contenitore una variabile membro di tipo puntatore (o un riferimento) ad un oggetto della classe contenuto
  - Implementando un costruttore della classe contenitore che riceva in ingresso un puntatore (un riferimento) ad un oggetto della classe contenuto



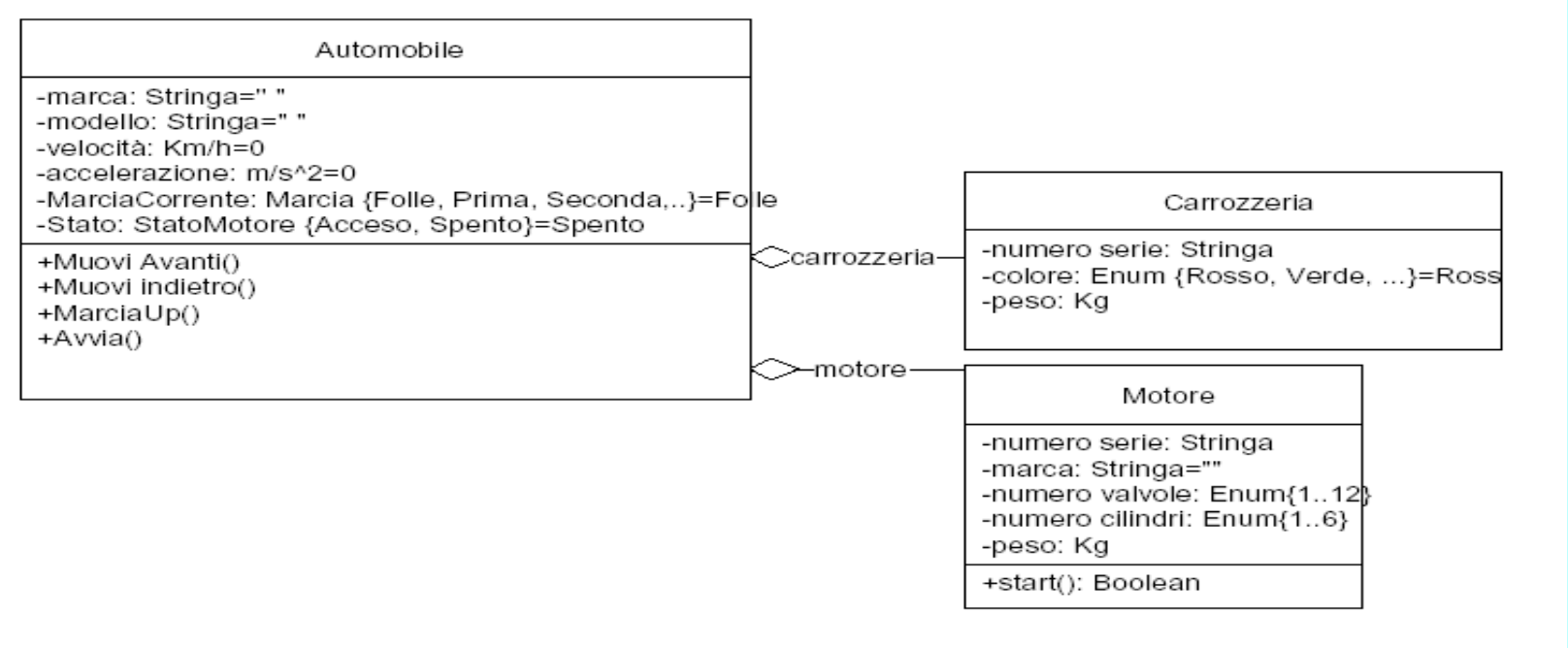
Codifica dell'aggregazione in C++ e in Java

In particolare, l'**aggregazione** si può realizzare in C++ o in Java come segue:

- In **C++**: il contenitore C ha una variabile membro privata di tipo **puntatore** ad un oggetto del tipo contenuto X
  - Il contenitore dovrà avere un costruttore che riceve in input un **puntatore** ad un oggetto di tipo X
- In **Java**: il contenitore C ha una variabile membro privata di tipo **riferimento** ad un oggetto del tipo contenuto X
  - Il contenitore dovrà avere un costruttore che riceve in input un **riferimento** ad un oggetto di tipo X

Esempio di aggregazione

- *Consideriamo il seguente esempio:*



Esso esprime il concetto che l'Automobile (ogni istanza di Automobile) ha una Carrozzeria ed un Motore.

La specifica prodotta indica un **contenimento debole (aggregazione).**

Volendo tradurre tale relazione in C++ (tramite l'uso dei puntatori), avremo:

Esempio di aggregazione

```
//file Automobile.h
#include "Motore.h"
#include "Carrozzeria.h"
class Automobile {
public:
  //il costruttore inizializza gli attributi propri e riceve un puntatore per ognuno degli oggetti aggregati
  //il costruttore inizializza Automobile anche in assenza di Motore e Carrozzeria
  Automobile(Stringa marcaI= "", Stringa modI="",…, Motore* motI=0, Carrozzeria* carI=0);
  ~Automobile();
  void Muovi_Avanti();
  void Muovi_indietro();
  Marcia MarciaUp();
  void Avvia();
private:
  Carrozzeria* carrozzeria;;        //traduzione aggregazione
  Motore* motore;                   //traduzione aggregazione
  Stringa marca;
  Stringa modello;
  Km_ora velocità;
  MetriSecondoQuadro accelerazione;
  Marcia marciacorrente;
  StatoMotore stato;
};
```

Esempio di aggregazione

- Notiamo l'implementazione del costruttore nel seguente frammento del file di implementazione della classe Automobile (Automobile.cpp)

```
//file Automobile.cpp
//Costruttore di Automobile
Automobile:: Automobile(Stringa marcaI, Stringa modI,…,Motore* motI, Carrozzeria* carI) :
// Inizializzazione variabili membro con lista di inizializzazione
marca(marcaI),  modello(modI), motore(motI),  carrozzeria(carI) {…};

//Implementazione funzione Avvia()
void Automobile::Avvia( ) {
  // accende il proprio motore ed inizializza la variabile membro stato
  stato = motore->start();
}
// Implementazione altre funzioni   .....
```

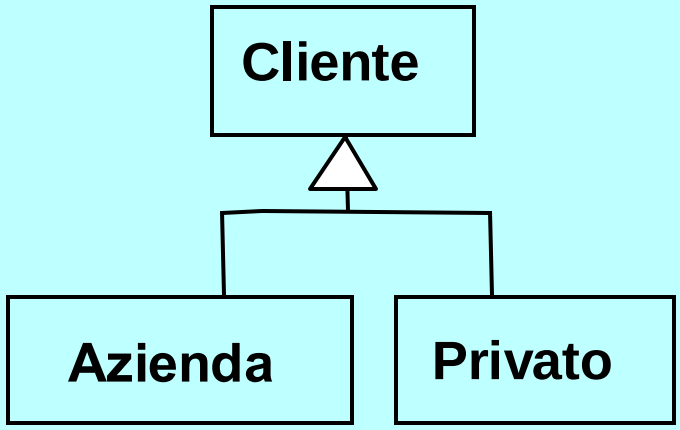
Esempio di aggregazione

- Il programma principale dovrà creare un oggetto di tipo Carrozzeria e Motore e poi l'oggetto di tipo Automobile. Ad es.:

```
#include "Automobile.h"
main () {                    //usa classe Automobile
  Motore m("AZ123","Ferrari", 6,12,1000); //definisce e inizializza oggetto di tipo Motore
  Carrozzeria c("12345ASA", "RossoFerrari",1500); //definisce e inizializza oggetto Carrozzeria

  Motore* puntatoreMotore=&m; //definisce e inizializza puntatore a motore
  Carrozzeria* puntatoreCarrozzeria=&c; //definisce e inizializza puntatore a carrozzeria

  //definisce e inizializza oggetto Automobile, fornendo puntatori
  Automobile auto1("Ferrari", …, puntatoreCarrozzeria, puntatoreMotore);

  auto1.Avvia(); //Avvia l'automobile (indirettamente è avviato il motore)

}    //al termine vengono separatamente distrutti m,c e auto1
```

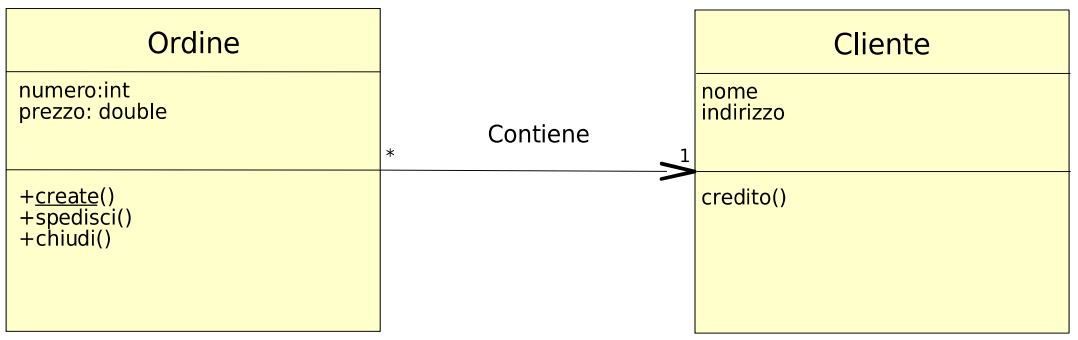
Traduzione di gerarchie di ereditarietà

- Ai fini della trasformazione in codice, il legame gen-spec si realizza mediante i meccanismi di ereditarietà previsti dal linguaggio di programmazione
  - In C++ esiste l'ereditarietà multipla
  - In Java esiste la parola chiave *extends*, e l'ereditarietà è singola (ma esistono le *interfaces*)



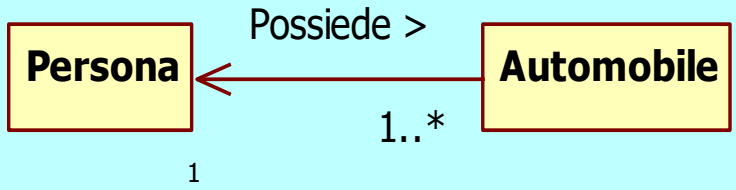
Traduzione delle associazioni

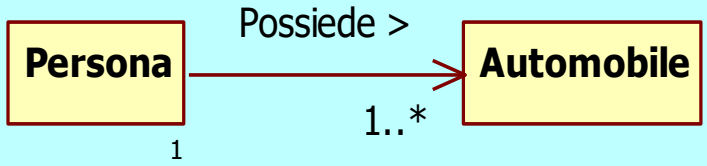
Le relazioni di associazione si traducono in C++ con una o più variabili membro di tipo puntatore alla classe associata (secondo la molteplicità dell'associazione)

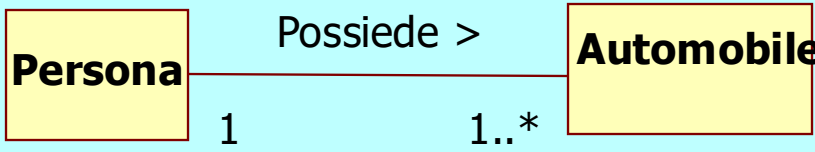


Traduzione delle associazioni

- **Molti a uno** (es.: una Persona, molte Automobili)
  - La classe Automobile ha un attributo Persona
  - oppure la classe Persona ha un attributo array di Automobile
  - o entrambe le cose
- La differenza tra le tre soluzioni è dettata dal verso di navigazione
- Si è parlato genericamente di array ma potrebbe trattarsi di altra struttura dati (es.: una lista)







Esempio C++

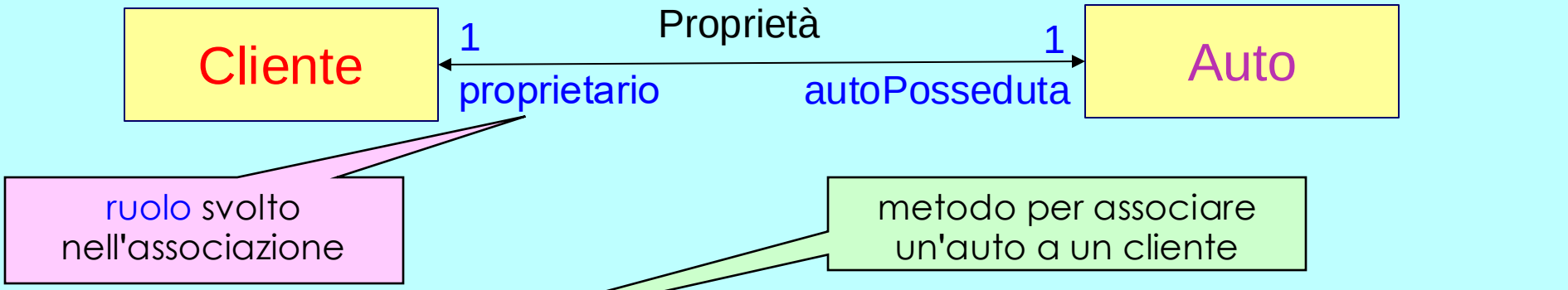
```
class A {
public: link(B* linkato):ruolo_di_B(linkato) {}
private: B* ruolo_di_B;
};

Class B {
public: link(A* linkato):ruolo_di_A(linkato) {}
private: A* ruolo_di_A;
};

Void main() {
A a;
B b;
a.link(&b);
b.link(&a);
};
```

Associazione uno a uno

```
class A {
public:     aggiungi(B* newobj);
            rimuovi(B* oldobj);
private:    lista_puntatori_a_B;
};

Class B {
public: link(A* linkato):ruolo_di_A(linkato) {}
private: A* ruolo_di_A;
};
```

Associazione uno a molti

Esempio codifica in Java associazione bidirez. uno-a-uno

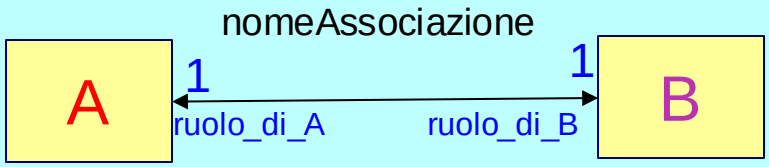



```
class Cliente {
  public associaAuto(Auto x) {this.autoPosseduta = x;}

  private Auto autoPosseduta;
};

class Auto {
  public associaProprietario(Cliente p)
            {this.proprietario=p;}

  private Cliente proprietario;
};
```

```
public static void main() {
  Cliente c;
  Auto a;

  c.associaAuto(a);
  a.associaProprietario(c);
};
```

N.B.: il ruolo nel modello UML diventa il nome della variabile membro

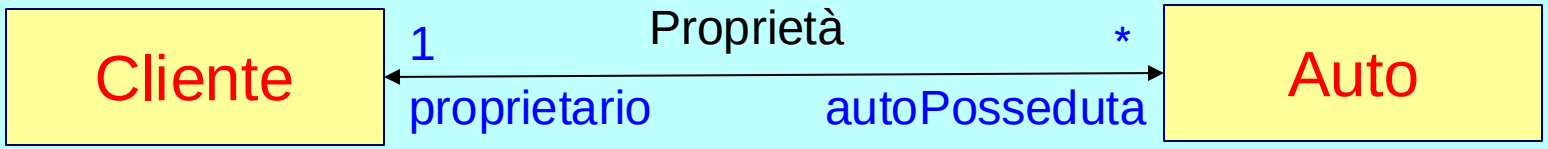
Codifica Java associazione bidirez. uno-a-uno (caso generico)




```
public static void main() {
 A a;
 B b;
 a.associa(b); //collega b -> a
 b.associa(a); //collega a -> b

 //scollega b -> a:
 a.rimuoviRuolo_di_B();
 //senza dimenticare di
 //scollegare a -> b !
  b.rimuoviRuolo_di_A();
};
```

```
class A {
   public aggiungi(B b) {this.ruolo_di_B = b;}
   public rimuoviRuolo_di_B()
                          {this.ruolo_di_B=NULL;}
   private B ruolo_di_B;
};
class B {
  public associa(A a) {this.ruolo_di_A = a;}
  public rimuoviRuolo_di_A
                         {this.ruolo_di_A = NULL;}
  private A ruolo_di_A;
};
```

Esempio codifica in Java associazione bidirez. uno-a-molti

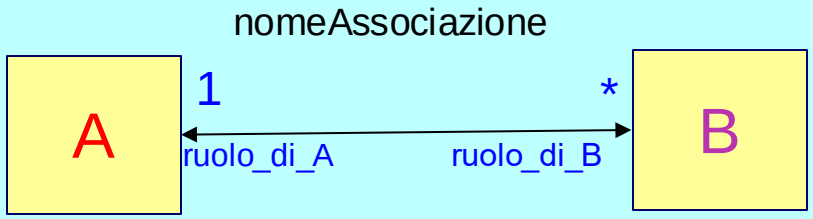


La classe Cliente ha come variabile membro un ArrayList di Auto.

```
class Cliente {
  public associaAuto(Auto x) {inserisci x in ArrayList}
  public rimuoviAuto(Auto x) {rimuovi x da ArrayList}
  private ArrayList<Auto> autoPosseduta;
};

class Auto {
  public associaProprietario(Cliente p)
            {this.proprietario=p;}
  public rimuoviProprietario() {this.proprietario=NULL;}
  private Cliente proprietario;
};
```

```
public static void main() {
  Cliente c;
  Auto a1, a2;

  c.associaAuto(a1);
  a1.associaProprietario(c);
  c.associaAuto(a2);
  a2.associaProprietario(c);
  …
  c.rimuoviAuto(a1);
  a1.rimuoviProprietario();
};
```

Codifica Java associazione bidirez. uno-a-molti (caso generico)




```
public static void main() {
  A a;
  B b1, b2;
  a.aggiungi(b1); //collega b1 -> a
  b1.associa(a);  //ed a -> b1
  a.aggiungi(b2); //collega b2 -> a
  b2.associa(a);  //ed a -> b2
  …
  a.rimuovi(b1); //scollega b1 -> a
  //non dimenticare di
  //scollegare a -> b1:
  b1.rimuoviRuolo_di_A();
};
```

```
class A {
   public aggiungi(B b) {inserisci b in ArrayList}
   public rimuovi(B b) {rimuovi b da ArrayList}
   private ArrayList<B> ruolo_di_B;
};
class B {
  public associa(A a) {this.ruolo_di_A = a;}
  public rimuoviRuolo_di_A()
          {this.ruolo_di_A = NULL;}
  private A ruolo_di_A;
};
```