Università di Roma "La Sapienza", Facoltà di Ingegneria

Corso di

PROGETTAZIONE DEL SOFTWARE

(Ing. Gestionale)

Prof. Giuseppe De Giacomo & Monica Scannapieco

Anno Accademico 2003/04

LA FASE DI PROGETTO E REALIZZAZIONE: DA UML A JAVA

Quarta parte

La fase di progetto e realizzazione

L'input di questa fase è costituito da:

- lo **schema concettuale**, formato da:
  - diagramma delle classi e degli oggetti
  - diagramma degli use case
  - diagramma degli stati e delle transizioni
- la **specifica**
  - una specifica per ogni classe
  - una specifica per ogni use case

Diagramma delle classi realizzativo

Il diagramma delle classi che si è costruito durante la fase di analisi, che chiameremo **diagramma delle classi concettuale**, è una concettualizzazione delle informazioni di interesse per la nostra applicazione che però prescinde da considerazioni tecnologiche.

Ora per potere tradurre tale diagramma in codice in modo efficace, dobbiamo innanzitutto prendere in esame alcuni aspetti su come la nostra applicazione utilizzerà le informazioni rappresentate dal diagramma.

In particolare dobbiamo decidere alcuni aspetti fondamentali per la realizzazione:

- Se gli attributi sono mutabili o immutabili

- Data una associazione, quali classi hanno responsabilità sulla stessa, nel senso che possono "navigare" (reperire le tuple) o aggiornare l'associazione (vedi dopo)

- Fornire ai clienti delle classi operazioni atte a verificare l'ammissibilità dello stato degli oggetti rispetto al diagramma delle classi concettuale (vedi dopo)

Tali decisioni sono formalizzate in un nuovo diagramma delle classi detto **diagramma delle classi realizzativo**.

Tale diagramma contiene tutte le informazioni necessarie alla realizzazione del codice, le uniche scelte che rimangono da fare sono scelte programmative.

Nota: il diagramma delle classi realizzativo è **peggiore** del diagramma delle classi concettuale, in quanto si sono effettuate delle scelte influenzate da tecnicalità realizzative, che servono per la realizzazione.

Traduzione in Java

Dobbiamo fare quanto segue:

1. **Traduzione dei tipi** (in genere, si usano tipi predefiniti o classi di libreria, ma in casi particolari si possono realizzare classi Java per rappresentare tipi particolari – vedi dopo).

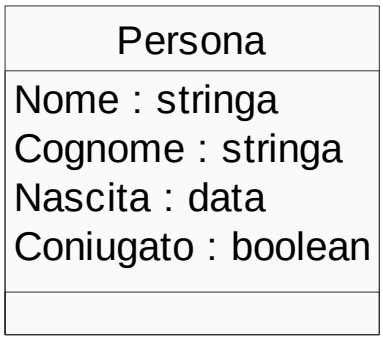
2. **Traduzione delle classi** (tipicamente, una classe Java per ogni classe UML).

3. Nel tradurre le classi, si procede alla:

   - **realizzazione delle associazioni**,
   - **realizzazione delle generalizzazioni**.

Singola classe UML con soli attributi

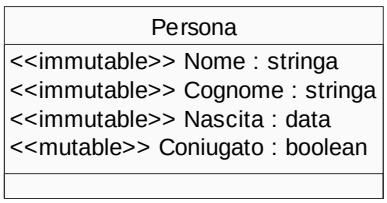
Consideriamo, inizialmente, diagrammi delle classi concettuali semplici: formati da una sola classe



Supponiamo che il nome, il cognome, e la data di nascita di una persona **non cambiano** mente l'essere coniugato **cambia**...

Diagramma delle classi realizzativo

... allora il diagramma delle classi realizzativo è il seguente:



Dove abbiamo esplicitato quali attributi non possono cambiare (**immutable**) e quali possono (**mutable**).

Una volta stabilito questo possiamo passare alla realizzazione del codice.

Realizzazione di classe UML con soli attributi

Per il momento, consideriamo il caso in cui la molteplicità di tutti gli attributi sia 1..1.

Per realizzare una classe UML *C* in termini di una classe Java `C`, le regole generali sono:

- La classe Java `C` è `public` e si trova in un file dal nome `C.java`.
- `C` è derivata da `Object` (no `extends`).
- La classe Java `C` avrà opportuni campi dati per gli attributi della classe UML *C*.
- La classe Java `C` avrà opportune funzioni (metodi) per gestire gli attributi e per costruire gli oggetti della classe.

Metodologia per la realizzazione: campi dati

I campi dati della classe Java `C` corrispondono agli attributi della classe UML *C*. Le regole principali sono le seguenti:

- I campi dati di `C` sono tutti `private` (o `protected`), per incrementare l'information hiding.

- Se i campi rappresentano un attributo **immutable** allora possono essere dichiarati `final` poiché non vengono più cambiati dopo la crezione dell'oggetto; altrimenti, non sono `final`.

- Si sceglie un opportuno valore iniziale per ogni attributo, o affidandosi al valore di default di Java o facendo in modo che il valore iniziale sia fissato, oggetto per oggetto, mediante un costruttore (vedi dopo).

Metodologia per la realizzazione: campi dati

- Ad un singolo attributo UML possono corrispondere uno o più campi dati (ad esempio, all'attributo *dataNascita*, possono corrispondere i tre campi `giorno`, `mese`, e `anno` della classe Java).

  I campi dati sono di un tipo base Java (`int`, `float`, ...) o `String`, ogni volta ci sia una chiara corrispondenza con il tipo dell'attributo della classe UML.

- Per due casi verranno dati maggiori dettagli in seguito:

  1. quando gli attributi UML hanno una loro molteplicità (ad es., *NumTel: int* {0..∗});

  2. quando non esiste in Java un tipo base o una classe predefinita che corrisponda chiaramente al tipo dell'attributo UML (ad es., *Valuta*).

Metodologia per la realizzazione: campi funzione

I campi funzione della classe `C` sono tutti `public`, e si classificano in:

Costruttori: devono inizializzare tutti i campi dati, esplicitamente o implicitamente.

Nel primo caso, le informazioni per l'inizializzazione vengono tipicamente acquisite tramite gli argomenti.

Funzioni `get`: per ogni campo dato (ad esempio `nome`), occorre definire una corrispondente funzione **get** (ad esempio `getNome()`), che serve a restituire al cliente, per ogni oggetto della classe, il valore dell'attributo.

Funzioni `set`: vanno previste solo per quei campi dati che possono mutare (cioè dichiarati **mutable**). Per ogni campo di questo tipo, la funzione **set** consente al cliente, per ogni oggetto della classe, di cambiare il valore dell'attributo. Ad esempio, `x.setEta(35)` fissa a 35 il valore del campo `eta` dell'oggetto `x`.

Metodologia: funzioni speciali

`equals()`: **non è opportuno** fare overriding della funzione `equals()` ereditata dalla classe `Object`.

Infatti due entità sono uguali solo se in realtà sono la stessa entità e quindi il comportamento di default della funzione `equals()` è corretto.

`clone()`: in molti casi, è ragionevole decidere di **non mettere a disposizione la possibilità di copiare un oggetto**, e non rendere disponibile la funzione `clone()` (non facendo overriding della funzione `protected` ereditata da `Object`).

Questa scelta deve essere fatta solo nel caso in cui si vuole che i moduli clienti utilizzino ogni oggetto della classe singolarmente e direttamente – *maggiori dettagli in seguito*.

`toString()`: si può prevedere di farne overriding, per avere una rappresentazione testuale dell'oggetto.

Realizzazione in Java della classe Persona

```
// File ParteQuarta/SoloAttributi/Persona.java

public class Persona {
  private final String nome, cognome;
  private final int giorno_nascita, mese_nascita, anno_nascita;
  private boolean coniugato;
  public Persona(String n, String c, int g, int m, int a) {
    nome = n;
    cognome = c;
    giorno_nascita = g;
    mese_nascita = m;
    anno_nascita = a;
  }
  public String getNome() {
    return nome;
  }
```



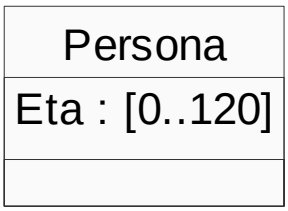
```
  public String getCognome() {
    return cognome;
  }
  public int getGiornoNascita() {
    return giorno_nascita;
  }
  public int getMeseNascita() {
    return mese_nascita;
  }
  public int getAnnoNascita() {
    return anno_nascita;
  }
  public void setConiugato(boolean c) {
    coniugato = c;
  }
  public boolean getConiugato() {
    return coniugato;
  }
```

```
  public String toString() {
    return nome + ' ' + cognome + ", " + giorno_nascita + "/" +
        mese_nascita + "/" + anno_nascita + ", " +
        (coniugato?"coniugato":"celibe");
  }
}
```

Il problema dei valori non ammessi

In alcuni casi, il tipo base Java usato per rappresentare il tipo di un attributo ha dei valori **non ammessi** per quest'ultimo.

Ad esempio, nella classe UML *Persona* potrebbe essere presente un attributo età, con valori interi ammessi compresi fra 0 e 120.



In tali casi si pone il problema di assicurare che i valori usati nei parametri attuali del costruttore di `Persona` e della funzione `setEta()` siano coerenti con l'intervallo. Un problema simile si ha quando un'operazione di una classe o di uno use case ha **precondizioni**.

Vedremo due possibili approcci alla soluzione di questo problema.

Verifica nel lato client

Con il primo approccio è sempre **il cliente** a doversi preoccupare che siano verificate le condizioni di ammissibilità.

```
// File ParteQuarta/Precondizioni/LatoClient/Persona.java

public class Persona {
  private int eta;
  public Persona(int e) { eta = e; }
  public int getEta() { return eta; }
  public void setEta(int e) { eta = e; }
  public String toString() {
    return " (" + eta + " anni)";
  }
}

// File ParteQuarta/Precondizioni/LatoClient/Client.java
```



```
public class Client {
  public static void main(String[] args) {
    Persona giovanni = null;
    boolean ok = false;
    while (!ok) {
      System.out.println("Inserisci eta'");
      int eta = InOut.readInt();
      if (eta >= 0 && eta <= 120) { // CONTROLLO PRECONDIZIONI
        giovanni = new Persona(eta);
        ok = true;
      }
    }
    System.out.println(giovanni);
  }
}
```

Problemi dell'approccio lato client

Con tale approccio, il cliente ha bisogno di un certo grado di conoscenza dei meccanismi di funzionamento della classe, il che potrebbe causare un **aumento dell'accoppiamento**.

Inoltre, il controllo delle precondizioni verrà duplicato in ognuno dei clienti, con **indebolimento dell'estendibilità e della modularità**.

Per questo motivo, un altro approccio tipico prevede che sia la classe a doversi preoccupare della verifica delle condizioni di ammissibilità (si tratta, in altre parole, di un approccio **lato server**).

In tale approccio, le funzioni della classe lanceranno un'eccezione nel caso in cui le condizioni non siano rispettate. Il cliente intercetterà tali eccezioni, e intraprenderà le opportune azioni.

Verifica nel lato server

In questo approccio, quindi:

- Va definita un'opportuna classe (derivata da `Exception` – o `RuntimeException`, se vogliamo che l'eccezione sia "unchecked") che rappresenta le eccezioni sulle precondizioni.

- Nella classe server, le funzioni devono lanciare (mediante il costrutto `throw`) eccezioni nel caso in cui le condizioni di ammissibilità non siano verificate.

- La classe client deve intercettare mediante il costrutto `try catch` (o rilanciare) l'eccezione, e prendere gli opportuni provvedimenti.

Verifica nel lato server: esempio

```
// File ParteQuarta/Precondizioni/LatoServer/EccezionePrecondizioni.java

public class EccezionePrecondizioni extends Exception {
  public EccezionePrecondizioni(String m) {
    super(m);
  }
  public EccezionePrecondizioni() {
    super("Si e' verificata una violazione delle precondizioni");
  }
}

// File ParteQuarta/Precondizioni/LatoServer/Persona.java

public class Persona {
  private int eta;
  public Persona(int e) throws EccezionePrecondizioni {
```



```
    if (e < 0 || e > 120)  // CONTROLLO PRECONDIZIONI
      throw new
        EccezionePrecondizioni("L'eta' deve essere compresa fra 0 e 120");
    eta = e;
  }
  public int getEta() { return eta; }
  public void setEta(int e) throws EccezionePrecondizioni {
    if (e < 0 || e > 120)  // CONTROLLO PRECONDIZIONI
      throw new EccezionePrecondizioni();
    eta = e;
  }
  public String toString() {
    return " (" + eta + " anni)";
  }
}

// File ParteQuarta/Precondizioni/LatoServer/Client.java
```

```
public class Client {
  public static void main(String[] args) {
    Persona giovanni = null;
    boolean ok = false;
    while (!ok) {
      System.out.println("Inserisci eta'");
      int eta = InOut.readInt();
      try {
        giovanni = new Persona(eta);
        ok = true;
      }
      catch (EccezionePrecondizioni e) {
        System.out.println(e);
      }
    }
    System.out.println(giovanni);
  }
}
```

Classe UML con attributi e operazioni

A quanto detto per il caso di classe con soli attributi si aggiunge:

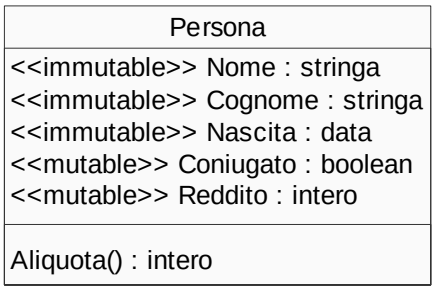
- Si analizza la specifica della classe UML *C*, che fornisce l'informazione sul significato di ogni operazione, cioè su quali sono le istruzioni che essa deve svolgere.

- Ogni operazione viene realizzata da una funzione `public` della classe Java.

  Sono possibili eventuali funzioni `private` o `protected` che dovessero servire per la realizzazione delle operazioni della classe UML *C*, ma che non vogliamo rendere disponibili ai clienti.

- Le precondizioni delle operazioni si trattano con l'approccio della verifica dal lato server, e quindi con opportune eccezioni.

Classe UML con attributi e operazioni: esempio



InizioSpecificaClasse Persona

Aliquota (): intero

pre: nessuna

post: *result* vale 0 se *this.Reddito* è inferiore a 5001, vale 20 se *this.Reddito* è compreso fra 5001 e 10000, vale 30 se *this.Reddito* è compreso fra 10001 e 30000, vale 40 se *this.Reddito* è superiore a 30000.

FineSpecifica

Realizzazione in Java

```
// File ParteQuarta/AttributiEOperazioni/Persona.java

public class Persona {
  private final String nome, cognome;
  private final int giorno_nascita, mese_nascita, anno_nascita;
  private boolean coniugato;
  private int reddito;
  public Persona(String n, String c, int g, int m, int a) {
    nome = n;
    cognome = c;
    giorno_nascita = g;
    mese_nascita = m;
    anno_nascita = a;
  }
  public String getNome() {
    return nome;
```

```
}
public String getCognome() {
  return cognome;
}
public int getGiornoNascita() {
  return giorno_nascita;
}
public int getMeseNascita() {
  return mese_nascita;
}
public int getAnnoNascita() {
  return anno_nascita;
}
public void setConiugato(boolean c) {
  coniugato = c;
}
public boolean getConiugato() {
  return coniugato;
```

```
}
public void setReddito(int r) {
  reddito = r;
}
public int getReddito() {
  return reddito;
}
public int aliquota() {
  if (reddito < 5001)
    return 0;
  else if (reddito < 10001)
    return 20;
  else if (reddito < 30001)
    return 30;
  else return 40;
}
public String toString() {
  return nome + ' ' + cognome + ", " + giorno_nascita + "/" +
```

```
        mese_nascita + "/" + anno_nascita + ", " +
        (coniugato?"coniugato":"celibe") + ", aliquota fiscale: " +
        aliquota();
  }
}
```

Esempio di cliente

InizioSpecificaUseCase Analisi Statistica

QuantiConiugati (*i: Insieme(Persona)): intero*
pre: nessuna
post: *result* è il numero di coniugati nell'insieme di persone *i*

FineSpecifica

Assumiamo per il momento di rappresentare l'insieme in input semplicemente come un vettore.

```
public class AnalisiStatistica { // ...
    public static int quantiConiugati(Persona[] vett) {
        int quanti = 0;
        for (int i = 0; i < vett.length; i++)
            if (vett[i].getConiugato())
                quanti++;
        return quanti;
    }
```

Esercizio 1: classi UML con operazioni

Realizzare in Java la classe UML *Persona* che comprende anche la funzione *Età*, così specificata:

InizioSpecificaClasse **Persona**

Aliquota (): intero . . .

Età (*d*: data): intero
pre: nessuna
post: *result* è l'età (in mesi) della persona *this* alla data *d*.

FineSpecifica

Soluzione esercizio 1

```
// File ParteQuarta/Esercizio1-2/Persona.java

public class Persona {
  private final String nome, cognome;
  private final int giorno_nascita, mese_nascita, anno_nascita;
  private boolean coniugato;
  private int reddito;
  public Persona(String n, String c, int g, int m, int a) {
    nome = n;
    cognome = c;
    giorno_nascita = g;
    mese_nascita = m;
    anno_nascita = a;
  }
  public String getNome() {
    return nome;
  }
  public String getCognome() {
    return cognome;
  }
  public int getGiornoNascita() {
    return giorno_nascita;
  }
  public int getMeseNascita() {
```

```
    return mese_nascita;
  }
  public int getAnnoNascita() {
    return anno_nascita;
  }
  public void setConiugato(boolean c) {
    coniugato = c;
  }
  public boolean getConiugato() {
    return coniugato;
  }
  public void setReddito(int r) {
    reddito = r;
  }
  public int getReddito() {
    return reddito;
  }
  public int aliquota() {
    if (reddito < 5001)
      return 0;
    else if (reddito < 10001)
      return 20;
    else if (reddito < 30001)
      return 30;
    else return 40;
  }
```

```
  public int eta(int g, int m, int a) {
    int mesi = (a - anno_nascita) * 12 + m - mese_nascita;
    if (!compiutoMese(g))
        mesi--;
    return mesi;
  }
  private boolean compiutoMese(int g) {
    return g >= giorno_nascita;
  }
  public String toString() {
    return nome + ' ' + cognome + ", " + giorno_nascita + "/" +
        mese_nascita + "/" + anno_nascita + ", " +
        (coniugato?"coniugato":"celibe") + ", aliquota fiscale: " +
        aliquota();
  }
}
```

Esercizio 2: cliente della classe

Realizzare in Java lo use case *Analisi Statistica* che comprende anche l'operazione *EtàMediaRicchi*:

InizioSpecificaUseCase Analisi Statistica

QuantiConiugati (*i: Insieme(Persona)): intero* . . .

EtàMediaRicchi (*i: Insieme(Persona), d: data): reale*
pre: *i* contiene almeno una persona
post: *result* è l'età media (in mesi) alla data *d* delle persone con aliquota massima tra le persone nell'insieme *i*

FineSpecifica

Rappresentare l'insieme in input semplicemente come un vettore.

Soluzione esercizio 2

```
// File AnalisiStatistica.java
public class AnalisiStatistica { // ...
  public static double etaMediaRicchi(Persona[] vett, int g, int m, int a) {
      int aliquotaMassima = 0;
      for (int i = 0; i < vett.length; i++)
          if (vett[i].aliquota() > aliquotaMassima)
              aliquotaMassima = vett[i].aliquota();
      int quantiRicchi = 0;
      double sommaEtaRicchi = 0.0;
      for (int i = 0; i < vett.length; i++)
          if (vett[i].aliquota() == aliquotaMassima) {
              sommaEtaRicchi += vett[i].eta(g,m,a);
              quantiRicchi++;
          }
      return sommaEtaRicchi / quantiRicchi;
  }
}
```

Realizzazione di una classe con un'associazione

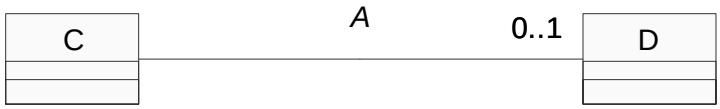
Nel realizzare una classe che è coinvolta in un'associazione, ci dobbiamo chiedere se la classe ha **responsabilità** sull'associazione.

Diciamo che **una classe C ha responsabilità sull'associazione A**, quando, per ogni oggetto x che è istanza di C vogliamo poter eseguire opportune operazioni sulle istanze di A a cui x partecipa, che hanno lo scopo di:

- conoscere l'istanza (o le istanze) di A alle quali x partecipa,
- aggiungere una nuova istanza di A alla quale x partecipa,
- cancellare una istanza di A alla quale x partecipa,
- aggiornare il valore di un attributo di una istanza di A alla quale x partecipa.

L'informazione su quali sono le classi che hanno responsabilità sulle varie associazioni si trova nella specifica delle classi. Per ogni associazione A, **deve** esserci almeno una delle classi coinvolte che ha responsabilità su `A`.

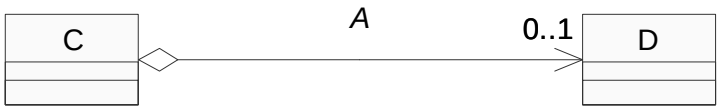
Realizzazione delle associazioni: primo caso



Consideriamo il caso in cui

- l'associazione sia binaria, di molteplicità 0..1 da C a D
- la specifica della classe C ci dice che essa è l'unica ad avere responsabilità sull'associazione A (cioè dobbiamo realizzare un "solo verso" della associazione)
- l'associazione A non abbia attributi
- non siamo interessati a mantenere esplicita la reppresentazione dei link nella associazione A.

Diagramma delle classi realizzativo



Nel diagramma realizzativo sostituiamo l'associazione con una **aggregazione** navigabile solo nella direzione da *C* a *D*. In questo modo modelliamo l'associazione come una relazione "has-a": gli oggetti della classe *C* hanno (come parte) un oggetto della classe *D*.

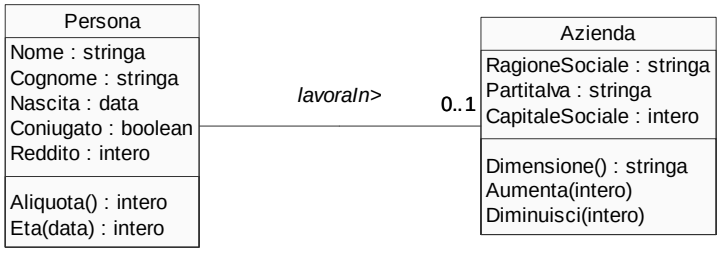
Realizzazione delle associazioni: primo caso

In questo caso, la realizzazione del codice è simile a quella per un attributo. Infatti, oltre a quanto stabilito per gli attributi e le operazioni, per ogni associazione *A* del tipo mostrato in figura, aggiungiamo alla classe Java `C`

- un campo dato di tipo `D` nella parte `private` (o `protected`) che rappresenta, per ogni oggetto `x` della classe `C`, l'oggetto della classe `D` connesso ad `x` tramite l'associazione `A`,
- una funzione `get` che consente di calcolare, per ogni oggetto `x` della classe `C`, l'oggetto della classe `D` connesso a `x` tramite l'associazione `A` (la funzione restituisce `null` se `x` non partecipa ad alcuna istanza di `A`),
- una funzione `set`, che consente di stabilire che l'oggetto `x` della classe `C` è legato ad un oggetto `y` della classe `D` tramite l'associazione `A` (sostituendo l'eventuale legame già presente); se tale funzione viene chiamata con `null` come argomento, allora la chiamata stabilisce che l'oggetto `x` della classe `C` non è più legato ad alcun oggetto della classe `D` tramite l'associazione `A`.

Due classi legate da associazione: esempio

La ragione sociale e la partita Iva di un'azienda **non cambiano**.



Supponiamo che solo *Persona* abbia responsabilità sull'associazione (non ci interessa conoscere i dipendenti di un'azienda, ma solo in quale azienda lavora una persona che lavora), assumiamo inoltre di non volere mantenere la rappresentazione esplicita dei link della associazione.

Specifica della classe UML **Azienda**

InizioSpecificaClasse Azienda

Dimensione (): *stringa*
pre: nessuna
post: *result* vale "Piccola" se *this.CapitaleSociale* è inferiore a 51, vale "Media" se *this.CapitaleSociale* è compreso fra 51 e 250, vale "Grande" se *this.CapitaleSociale* è superiore a 250

Aumenta (*i: intero*)
pre: $i > 0$
post: *this.CapitaleSociale* vale *pre(this.CapitaleSociale)* + *i*

Diminuisci (*i: intero*)
pre: $1 \leq i \leq$ *this.CapitaleSociale*
post: *this.CapitaleSociale* vale *pre(this.CapitaleSociale)* - *i*

FineSpecifica

Diagramma delle classi realizzativo

La ragione sociale e la partita Iva di un'azienda **non cambiano**.

Supponiamo che solo *Persona* abbia responsabilità sull'associazione (non ci interessa conoscere i dipendenti di un'azienda, ma solo in quale azienda lavora una persona che lavora), assumiamo inoltre di non volere mantenere la rappresentazione esplicita dei link della associazione.

Realizzazione in Java della classe Azienda

```
// File ParteQuarta/Associazioni01/Azienda.java

public class Azienda {
  private final String ragioneSociale, partitaIva;
  private int capitaleSociale;
  public Azienda(String r, String p) {
    ragioneSociale = r;
    partitaIva = p;
  }
  public String getRagioneSociale() {
    return ragioneSociale;
  }
  public String getPartitaIva() {
    return partitaIva;
  }
  public int getCapitaleSociale() {
```

```
    return capitaleSociale;
  }
  public void aumenta(int i) {
    capitaleSociale += i;
  }
  public void diminuisci(int i) {
    capitaleSociale -= i;
  }
  public String dimensione() {
    if (capitaleSociale < 51)
      return "Piccola";
    else if (capitaleSociale < 251)
      return "Media";
    else return "Grande";
  }
  public String toString() {
    return ragioneSociale + " (P.I.: " + partitaIva +
    "), capitale sociale: " + getCapitaleSociale() +
```

```
    ", tipo azienda: " + dimensione();
  }
}
```

Realizzazione in Java della classe Persona

```
public class Persona {
  // altri campi dati e funzione
  private Azienda lavoraIn;

  public Azienda getLavoraIn() {
    return lavoraIn;
  }
  public void setLavoraIn(Azienda a) {
    lavoraIn = a;
  }
  public String toString() {
    return nome + ' ' + cognome + ", " + giorno_nascita + "/" +
      mese_nascita + "/" + anno_nascita + ", " +
      (coniugato?"coniugato":"celibe") + ", aliquota fiscale: " + aliquota()
      (lavoraIn != null?", lavora presso la ditta " + lavoraIn:
       ", disoccupato");
  }
}
```


Esercizio 3: cliente

Realizzare in Java lo use case *Analisi Statistica* che comprende anche l'operazione *RedditoMedioInGrandiAziende*:

InizioSpecificaUseCase Analisi Statistica

RedditoMedioInGrandiAziende (*i: Insieme(Persona)): reale*
pre: *i* contiene almeno una persona che lavora in una grande azienda
post: *result* è il reddito medio delle persone che lavorano in una grande azienda nell'insieme di persone *i*

FineSpecifica

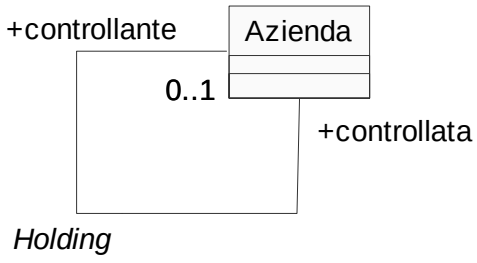
Rappresentare l'insieme in input semplicemente come un vettore.

Soluzione esercizio 3

```
// File ParteQuarta/Associazioni01/Esercizio3.java

public class Esercizio3 {
  public static double redditoMedioInGrandiAziende(Persona[] vett) {
    int quantiInGrandiAziende = 0;
    double sommaRedditoDipendentiGrandiAziende = 0.0;
    for (int i = 0; i < vett.length; i++)
      if (vett[i].getLavoraIn() != null &&
          vett[i].getLavoraIn().dimensione().equals("Grande")) {
        quantiInGrandiAziende++;
        sommaRedditoDipendentiGrandiAziende += vett[i].getReddito();
      }
    return sommaRedditoDipendentiGrandiAziende / quantiInGrandiAziende;
  }
}
```

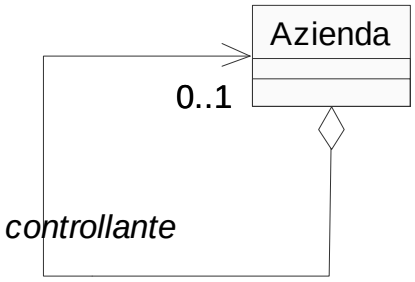

Realizzazione delle associazioni: secondo caso

Quanto detto vale anche per il caso in cui l'associazione coinvolga più volte la stessa classe. In questo caso il concetto di responsabilità si attribuisce ai ruoli, piuttosto che alla classe.



Supponiamo che la classe *Azienda* abbia la responsabilità su *holding*, solo nel ruolo *controllata*. Questo significa che, dato un oggetto x della classe *Azienda*, vogliamo poter eseguire operazioni su x per conoscere l'azienda controllante, per aggiornare l'azienda controllante, ecc. Supponiamo inoltre di non volere mantenere una rappresentazione esplicita dei link della associazione.

Diagramma delle classi realizzativo: secondo caso



Si noti che abbiamo sostituito l'associazione *holding* con l'aggregazione *controllante*.

In generale il nome della aggregazione viene scelto uguale al nome del ruolo (nell'esempio, il nome è `controllante`).

Realizzazione del codice: secondo caso

```
public class Azienda {
  // eventuali attributi .....
  private Azienda controllante; // il nome del campo e' uguale al ruolo
  // altre funzioni ....
  public Azienda getControllante() {
    return controllante;
  }
  public void setControllante(Azienda a) {
    controllante = a;
  }
}
```

Realizzazione delle associazioni: terzo caso

Consideriamo il caso in cui la classe *C* sia l'unica ad avere la responsabilità sull'associazione *A*, e vogliamo mantenere una **reppresentazione dei link** della associazione.

La necessità di rappresentare i link è tipicamente dovuto al fatto che l'associazione *A* abbia **attributi**.

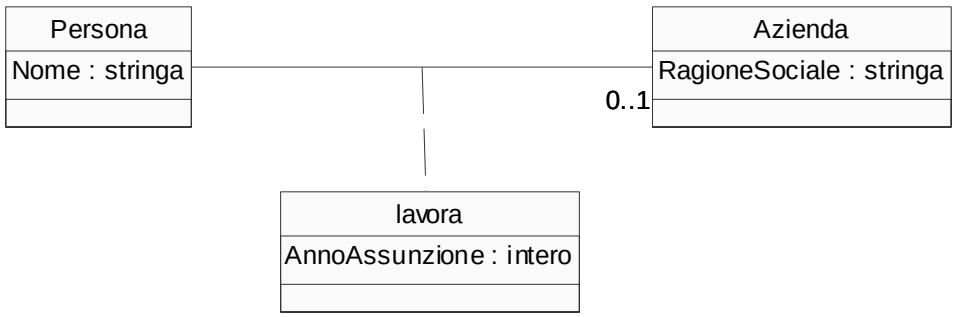
Assumiamo, per ora, che l'associazione abbia uno o più attrivuti di di molteplicità 1..1.

Rimangono le seguenti assunzioni:

- molteplicità 0..1;
- solo una delle due classi *ha responsabilità* sull'associazione (dobbiamo rappresentare **un solo verso** dell'associazione).

Realizzazione delle associazioni: terzo caso

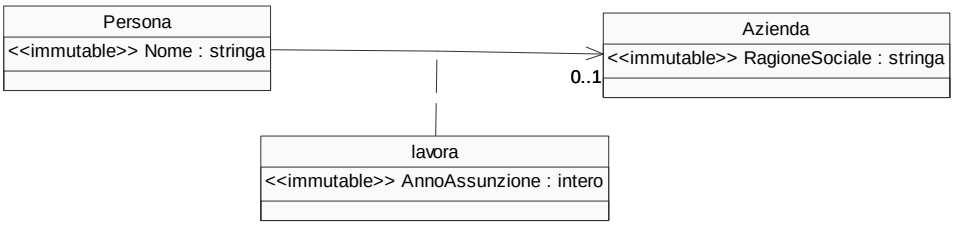
Esempio



Solo *Persona* ha responsabilità sull'associazione, vogliamo mantenere una rappresentazione esplicita dei link per gestire opportunamente gli attributi dell'associazione.

Diagramma delle classi realizzativo: terzo caso

Esempio



Nel diagramma delle classi realizzativo abbiamo mantenuto l'associazione, tuttavia ne abbiamo ristretto la navigabilità esplicitando che il verso di navigazione che interessa è quello da *Persona* ad *Azienda*.

In questo modo esprimiamo che nella realizzazione del codice manterremo una rappresentazione esplicita dei link dell'associazione, ma ne permetteremo di navigare l'associazione in un'unica direzione.

Attributi di associazione (cont.)

Per rappresentare l'associazione *As* fra le classi UML *A* e *B* con attributi, introduciamo **un'ulteriore classe** Java `TipoLinkAs`, che ha lo scopo di rappresentare i link fra gli oggetti delle classi *A* e *B*. In particolare, ogni link (presente al livello estensionale) fra un oggetto di classe *A* ed uno di classe *B* sarà rappresentato da un oggetto di classe `TipoLinkAs`.

La classe Java `TipoLinkAs` avrà campi dati per rappresentare:

- gli attributi dell'associazione;
- i riferimenti agli oggetti delle classi `A` e `B` relativi al link.

La classe Java `TipoLinkAs` avrà inoltre delle funzioni che consentono di gestire i suoi campi dati (costruttore, funzioni **get**), e la funzione `equals` per verificare l'uguaglianza solo sugli oggetti collegati dal link, ignorando gli attributi.

Attributi di associazione (cont.)

Supponendo che solo la classe UML *A* abbia responsabilità sull'associazione *As*, la classe Java `A` che la realizza dovrà tenere conto della presenza dei link.

Quindi, la classe Java `A` avrà:

- un campo dati di tipo `TipoLinkAs`, per rappresentare l'eventuale link;

  in particolare, se tale campo vale `null`, allora significa che l'oggetto di classe `A` non è associato ad un oggetto di classe `B`;

- opportuni campi funzione che permettono di gestire il link (funzioni `get`, `inserisci`, `elimina`).

Realizzazione in Java della classe TipoLinkLavora

```
// File ParteQuarta/Ass01Attr/TipoLinkLavora.java

public class TipoLinkLavora {
  private final Persona laPersona;
  private final Azienda laAzienda;
  private final int annoAssunzione;

  public TipoLinkLavora(Azienda x, Persona y, int a) {
    laAzienda = x; laPersona = y; annoAssunzione = a;
  }

  public Azienda getAzienda() { return laAzienda; }
  public Persona getPersona() { return laPersona; }
  public int getAnnoAssunzione() { return annoAssunzione; }

  public boolean equals(Object o) {
```

```
    if (o != null && getClass().equals(o.getClass())) {
      TipoLinkLavora b = (TipoLinkLavora)o;
      return b.laPersona != null && b.laAzienda != null &&
             b.laPersona == laPersona && b.laAzienda == laAzienda;
    }
    else return false;
  }
}
```

Realizzazione in Java della classe Persona

```
// File ParteQuarta/Ass01Attr/Persona.java

public class Persona {
  private final String nome;
  private TipoLinkLavora link;
  public Persona(String n) { nome = n; }
  public String getNome() { return nome; }
  public void inserisciLinkLavora(TipoLinkLavora t) {
    if (link == null && t != null &&
        t.getAzienda() != null && t.getPersona() == this)
      link = t;
  }
  public void eliminaLinkLavora() {
    link = null;
  }
  public TipoLinkLavora getLinkLavora() { return link;  }
}
```

Considerazioni sulle classi Java

- Si noti che i campi dati nella classe `TipoLinkLavora` sono tutti `final`.

  Di fatto un oggetto della classe è *immutabile*, ovvero una volta creato non può più essere cambiato.

- La funzione `inserisciLinkLavora()` della classe `Persona` deve assicurarsi che:

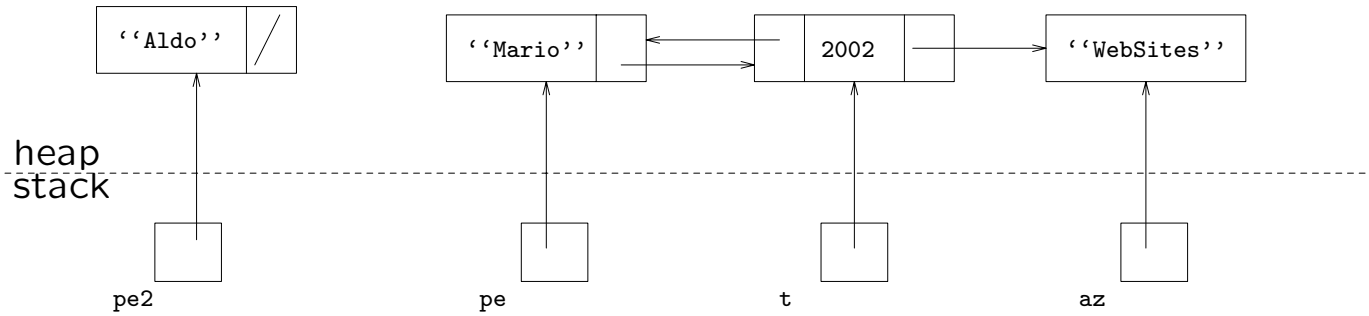
  – la persona oggetto di invocazione non partecipi già ad un link della associazione *lavora*;

  – l'oggetto che rappresenta il link esista;

  – l'azienda a cui si riferisce il link esista;

  – la persona a cui si riferisce il link sia l'oggetto di invocazione.

Possibile stato della memoria



Due oggetti di classe `Persona`, di cui uno che lavora ed uno no.

Realizzazione della situazione di esempio

```
public static void main (String[] args) {

  Azienda az = new Azienda("WebSites");
  Persona pe = new Persona("Mario");
  Persona pe2 = new Persona("Aldo");

  TipoLinkLavora t = new TipoLinkLavora(az,pe,2002);

  pe.inserisciLinkLavora(t);
}
```


Esercizio 4: cliente

Realizzare in Java lo use case *Ristrutturazione Industriale*:

InizioSpecificaUseCase **Ristrutturazione Industriale**

AssunzioneInBlocco *(i: Insieme(Persona), a: Azienda, an: intero)*
pre: nessuna
post: tutte le persone nell'insieme di persone *i* vengono assunte dall'azienda *a* nell'anno *an*

AssunzionePersonaleEsperto *(i: Insieme(Persona), a: Azienda, av: intero, an: intero)*
pre: nessuna
post: tutte le persone nell'insieme di persone *i* che lavorano in un'azienda qualsiasi fin dall'anno *av* vengono assunte dall'azienda *a* nell'anno *an*

FineSpecifica

Rappresentare l'insieme in input semplicemente come un vettore.

Soluzione esercizio 4

```
// File ParteQuarta/Ass01Attr/Esercizio4.java

public class Esercizio4 {
  public static void assunzioneInBlocco(Persona[] vett, Azienda az, int an) {
    for (int i = 0; i < vett.length; i++) {
      vett[i].eliminaLinkLavora();
      TipoLinkLavora temp = new TipoLinkLavora(az,vett[i],an);
      vett[i].inserisciLinkLavora(temp);
    }
  }
  public static void assunzionePersonaleEsperto(Persona[] vett,
                                                Azienda az, int av, int an) {
    for (int i = 0; i < vett.length; i++) {
      if (vett[i].getLinkLavora() != null &&
          vett[i].getLinkLavora().getAnnoAssunzione() <= av) {
        vett[i].eliminaLinkLavora();
        TipoLinkLavora temp = new TipoLinkLavora(az,vett[i],an);
        vett[i].inserisciLinkLavora(temp);
      }
    }
  }
}
```


Metodi naive di realizzazione

Se abbiamo una associazione con attributi ma non siamo interessati alla rappresentazione esplicita dei link possimo realizzare tale associazione in modo utilizzando un campo dati per l'associazione e un campo dati addizionale per ciascuno degli attributi. Il risultato però è che tale realizzazione va fatta ad-hoc caso per caso, facendo uno sforzo per far si che il campo dati che rappresenta (implicitamente) l'associazione e quelli che rappresentano i suoi attributi mantengano informazioni allineate.

Consideriamo l'esempio discusso in precedenza

```
public class Persona {
  // eventuali attributi .....
  private Azienda lavora;
  private int annoAssunzione;
  // altre funzioni ....
  public Azienda getLavora() { return lavora; }
```

```
  public int getAnnoAssunzione() {
    if (a=null) throws new RuntimeException(‘‘valore non significativo’’);
    else return annoAssunzione;
  }
  public void setLavora(Azienda a, int x) {
    if (a != null) { lavora = a; annoAssunzione = x; }
  }
  public void eliminaLavora() { lavora = null; }
}
```

Osservazioni sul metodo naive

La funzione `SetLavora()` ha ora due parametri, perché nel momento in cui si lega un oggetto della classe `C` ad un oggetto della classe `D` tramite `A`, occorre specificare anche il valore dell'attributo dell'associazione (essendo tale attributo di tipo 1..1).

Il cliente della classe ha la responsabilità di chiamare la funzione `getAnnoAssunzione()` correttamente, cioè quando l'oggetto di invocazione `x` effettivamente partecipa ad una istanza della associazione `lavora` (`x.getLavora() != null`).

Il fatto che l'attributo dell'associazione venga realizzato attraverso un campo dato della classe `C` non deve trarre in inganno: concettualmente l'attributo appartiene all'associazione, ma è evidente che, essendo l'associazione 0..1 da `C` a `D`, ed essendo l'attributo di tipo 1..1, dato un oggetto `x` di `C` che partecipa all'associazione `A`, associato ad `x` c'è uno ed un solo valore per l'attributo. Quindi è corretto, in fase di implementazione, attribuire alla classe `C` il campo dato che rappresenta l'attributo dell'associazione.

Quarto caso: associazioni binarie di tipo 0..*

Ci concentriamo ora su associazioni binarie **con molteplicità 0..***.

Rimangono le seguenti assunzioni:

- senza attributi di associazione (solo per semplicità, in realtà li sappiamo già trattare);
- solo una delle due classi *ha responsabilità* sull'associazione (dobbiamo rappresentare **un solo verso** dell'associazione).

Gli altri casi verranno considerati in seguito.

Associazioni binarie, molteplicità 0..* (cont.)

Per rappresentare l'associazione *As* fra le classi UML *A* e *B* con molteplicità 0..* abbiamo bisogno di una **struttura di dati** per rappresentare i link fra un oggetto di classe *A* e più oggetti di classe *B*.

In particolare, la classe Java `A` avrà:

- un campo dati di un tipo opportuno (ad esempio `Set`, o `InsiemeSS`, vedi dopo), per rappresentare i link;
- opportuni campi funzione che permettano di gestire il link (funzioni `get`, `inserisci`, `elimina`).

Strutture di dati

In questo corso non vengono forniti algoritmi e metodologie per la gestione di strutture di dati (esiste il corso di *Algoritmi e strutture di dati*).

Abbiamo tuttavia la necessità di organizzare e gestire **riferimenti** secondo determinate politiche. In particolare faremo riferimento ad alcuni noti *tipi astratti*, quali:

- *Insieme*, astrazione di strutture *non ordinate*;
- *Vettore*, astrazione di strutture *di dimensione limitata a priori e ordinate rispetto ad un indice intero*;
- *Pila*, astrazione di strutture *gestite con una politica LIFO*;
- *Coda*, astrazione di strutture *gestite con una politica FIFO*.

Strutture di dati (cont.)

Java fornisce classi predefinite che realizzano alcuni di questi tipi di dato (ad es. `HashSet`).

Tuttavia queste classi permettono l'inserimento di riferimenti di tipo `Object` **senza controllarne l'omogeneità**. Di fatto lasciano al cliente l'onere di controllare che non vengano inseriti riferimenti disomogenei (di classi diverse).

Per questo motivo, utilizzeremo alcune classi "collezione" Java **realizzate appositamente**. Di tali classi:

- **non siamo tenuti** a conoscere i dettagli realizzativi;
- siamo tenuti a conoscere solamente la "API" (*Application Programmer's Interface*), ovvero i servizi forniti.

Classi collezione

Le classi collezione proposte hanno le seguenti caratteristiche:

1. Sono **generiche**, nel senso che permettono di gestire collezioni di elementi di un tipo qualunque, ma **omogenee** (ovvero di elementi che sono tutti istanze di una stessa classe). In particolare:

   - La classe alla quale devono appartenere gli elementi deve essere passata al costruttore della classe collezione, tramite un riferimento alla classe predefinita `Class`.

     Ricordiamo che, per ogni classe `C` (predefinita o no), l'espressione `C.class` denota un riferimento ad un oggetto della classe `Class` che identifica univocamente `C`.

   - La classe collezione consente l'inserimento di riferimenti di tipo `Object`.

   - Tuttavia, se il tipo del riferimento non è compatibile con la classe comunicata al costruttore, l'inserimento non ha luogo.

Classi collezione (cont.)

2. Fanno l'overriding delle funzioni speciali

   - `equals()`, che verifica l'uguaglianza profonda;

   - `clone()`, che effettua la copia profonda;

   - `toString()`, che associa una stringa stampabile all'oggetto di invocazione.

Implementazione dell'interfaccia Set

- La classe `InsiemeArray` (`InsiemeLista`), vista nella parte 1 del corso, implementa l'interfaccia predefinita `Set` del Collections Framework e realizza il *tipo astratto Insieme*.

- La classe `InsiemeArrayOmogeneo` (`InsiemeListaOmogeneo`) specializza la classe `InsiemeArray` (`InsiemeLista`), imponendo l'omogeneità degli elementi dell'insieme.

Noi nel seguito faremo tipicamente uso di `Set` e `InsiemeArrayOmogeneo` per rappresentare collezioni.

La classe InsiemeSS

La classe `InsiemeSS` realizza il *tipo astratto Insieme* ed è una alternativa a `Set`/`InsiemeArrayOmogeneo` utilizzata in alcuni esercizi d'esame degli anni passati. . L'interfaccia pubblica (API) di `InsiemeSS` è costituita dalle seguenti funzioni:

```
public class InsiemeSS implements Cloneable {
  public InsiemeSS(Class cl)
  public boolean estVuoto()
  public boolean membro(Object e) // verifica se nell'insieme e' presente un
                                  // elemento uguale (mediante equals) ad e
  public void inserisci(Object e) // inserisce solo se nell'insieme non e'
                                  // gia' presente un elemento uguale
                                  // (mediante equals) ad e
  public void elimina(Object e)
  public Object scegli()
  public int cardinalita()
```

```
    // funzioni speciali ereditate da Object
  public boolean equals(Object o)
  public Object clone()
  public String toString()
}
```

Una possibile realizzazione della classe Insieme

```
// File ParteQuarta/Insieme/InsiemeSS.java

public class InsiemeSS implements Cloneable {
  // funzioni del tipo astratto
  public InsiemeSS(Class cl) {
    inizio = null;
    cardinalita = 0;
    elemClass = cl;
  }
  public boolean estVuoto() {
    return inizio == null;
  }
  public boolean membro(Object e) {
    if (!elemClass.isInstance(e)) return false;
    else return appartiene(e,inizio);
  }
```

```
public void inserisci(Object e) {
  if (!elemClass.isInstance(e)) return;
  else if (appartiene(e,inizio)) return;
  else {
    Lista l = new Lista();
    l.info = e;
    l.next = inizio;
    inizio = l;
    cardinalita = cardinalita + 1;
  }
}
public void elimina(Object e) {
  if (!elemClass.isInstance(e)) return;
  if (!appartiene(e,inizio)) return;
  else {
    inizio = cancella(e,inizio);
    cardinalita = cardinalita - 1;
  }
```

```
}
public Object scegli() {
  if (inizio == null) return null;
  else return inizio.info;
}
public int cardinalita() {
  return cardinalita;
}
// funzioni speciali ereditate da Object
public boolean equals(Object o) {
  if (o != null && getClass().equals(o.getClass())) {
    InsiemeSS ins = (InsiemeSS)o;
    if (!elemClass.equals(ins.elemClass)) return false;
    // ins non e' un insieme del tipo voluto
    else if (cardinalita != ins.cardinalita) return false;
    // ins non ha la cardinalita' giusta
    else {
      // verifica che gli elementi nella lista siano gli stessi
```

```
      Lista l = inizio;
      while (l != null) {
        if (!appartiene(l.info,ins.inizio))
          return false;
        l = l.next;
      }
      return true;
    }
  }
  else return false;
}
public Object clone() {
  try {
    InsiemeSS ins = (InsiemeSS) super.clone();
    // chiamata a clone() di Object che esegue la copia campo a campo;
    // questa copia e' sufficiente per i campi cardinalita e elemClass
    // ma non per il campo inizio del quale va fatta una copia profonda
    ins.inizio = copia(inizio);
```

```
    return ins;
  } catch(CloneNotSupportedException e) {
    // non puo' accadere perche' implementiamo l'interfaccia cloneable,
    // ma va comunque gestita
    throw new InternalError(e.toString());
  }
}
public String toString() {
  String s = "{";
  Lista l = inizio;
  while (l != null) {
    s = s + l.info + " ";
    l = l.next;
  }
  s = s + "}";
  return s;
}
```

```
  // campi dati
  protected static class Lista {
    Object info;
    Lista next;
  }
  protected Lista inizio;
  protected int cardinalita;
  protected Class elemClass;

  // funzioni ausiliarie
  protected static boolean appartiene(Object e, Lista l){
    return (l != null) && (l.info.equals(e) || appartiene(e,l.next));
  }
  protected static Lista copia (Lista l) {
    if (l == null) return null;
    else {
      Lista ll = new Lista();
      ll.info = l.info;
```

```
      ll.next = copia(l.next);
      return ll;
    }
  }
  protected static Lista cancella(Object e, Lista l) {
    if (l == null) return null;
    else  if (l.info.equals(e)) return l.next;
    else {
      l.next = cancella(e,l.next);
      return l;
    }
  }
}
```

Commenti sulla classe InsiemeSS

- Il costruttore riceve un argomento di tipo `Class` che denota il tipo degli elementi. L'insieme costruito è **l'insieme vuoto**.
- Le funzioni `inserisci()` ed `elimina` fanno **side-effect sull'oggetto di invocazione**.
- Le rimanenti funzioni non effettuano side-effect sull'oggetto di invocazione.
- La funzione `scegli()` restituisce **uno qualsiasi** fra gli elementi dell'oggetto di invocazione (che deve essere non vuoto).
- È stato fatto l'overriding delle funzioni speciali.
- La funzione `cardinalita()` è **ridondante**.
- La classe `Lista` è interna alla classe `InsiemeSS`.

Clienti della classe InsiemeSS

Per dimostrare che la funzione `cardinalita()` è ridondante, mostriamo una funzione cliente (esterna alla classe `InsiemeSS`) che calcola la cardinalità dell'insieme passato come argomento.

```
public static int cardIter(InsiemeSS ins) {
    InsiemeSS copia = (InsiemeSS) ins.clone();
    int card = 0;
    while (!copia.estVuoto()) {
        Object elem = copia.scegli();
        card++;
        copia.elimina(elem);
    }
    return card;
}
```

Notiamo che la funzione cliente **fa una copia locale** dell'argomento.

Esercizio: clienti di InsiemeSS

Realizzare funzioni cliente (esterne alla classe `InsiemeSS`) per calcolare:

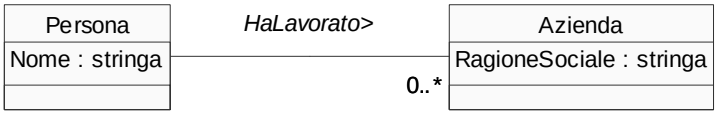
- la cardinalità dell'insieme passato come argomento mediante un algoritmo **ricorsivo**;
- l'insieme **unione** dei due insiemi passati come argomento;
- l'insieme **intersezione** dei due insiemi passati come argomento;
- l'insieme **differenza simmetrica** dei due insiemi passati come argomento.

Soluzione esercizio clienti InsiemeSS: cardinalità (ricorsiva)

```
public static int cardRic(InsiemeSS ins) {
    if (ins.estVuoto())
        return 0;
    else {
        Object elem = ins.scegli();
        ins.elimina(elem);
        int temp = cardRic(ins);
        ins.inserisci(elem);
        return temp + 1;
    }
}
```

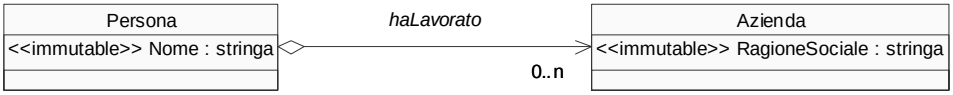
Si noti che non è necessaria copia dell'argomento.

Molteplicità 0..*: esempio



Supponiamo che solo *Persona* abbia responsabilità sull'associazione (non ci interessa conoscere i dipendenti passati di un'azienda, ma solo in quale azienda ha lavorato una persona).

Diagramma Realizzativo: Quarto Caso



Realizzazione in Java della classe Persona

```
// File ParteQuarta/AssOSTAR/Persona.java
import java.util.*;
import insiemearray.*;

public class Persona {
  private final String nome;
  private Set insieme_aziende;
  public Persona(String n) {
    nome = n;
    insieme_aziende = new InsiemeArrayOmogeneo(Azienda.class);
  }
  public String getNome() { return nome; }
  public void inserisciHaLavorato(Azienda az) {
    if (az != null) insieme_aziende.add(az);
  }
  public void eliminaHaLavorato(Azienda az) {
```



```
    if (az != null) insieme_aziende.remove(az);
  }
  public Iterator getHaLavorato() {
    return insieme_aziende.iterator();
  }
}
```

Classe Java Persona: considerazioni

- La classe ha un campo dati di tipo `Set`.

- Il costruttore della classe `Persona` crea un oggetto della classe `InsiemeArrayOmog` passando al costruttore come argomento il tipo degli elementi dell'insieme, cioè `Azienda.class`. Di fatto, viene creato un insieme vuoto di riferimenti di tipo `Azienda`.

- Ci sono varie funzioni che permettono di gestire l'insieme:
  - `inserisciHaLavorato(Azienda)`: permette di inserire un nuovo link;
  - `eliminaHaLavorato(Azienda)`: permette di eliminare un link esistente;
  - `getHaLavorato()`: permette di ottenere tutti i link di una persona, enumerandoli attraverso un iteratore.

Classe Java Persona: considerazioni (cont.)

- Si noti che la funzione `getHaLavorato()` non restituisce la struttura dati denotata da `insieme_link` ma un iteratore sulla stessa.

- Se così non fosse, daremmo al cliente della classe `Persona` la possibilità di modificare i link dell'associazione *HaLavorato* a suo piacimento, distruggendo la modularizzazione.

- Al contrario, i link dell'associazione *HaLavorato* devono essere gestiti solamente dalla classe `Persona`, che ha responsabilità sull'associazione.

Esempio di cliente

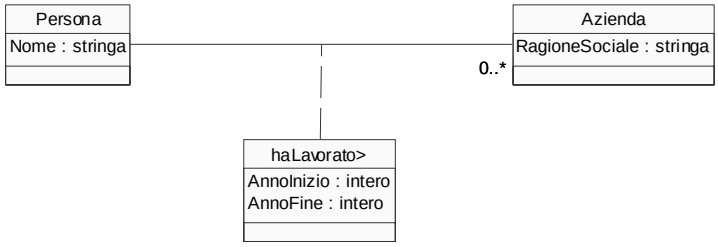
La seguente funzione stampa la ragione sociale di tutte le aziende per cui una certa persona ha lavorato.

```
public static void stampaAziende(Persona p) {
  System.out.println("La persona " + p.getNome() +
                     " ha lavorato nelle seguenti aziende:");
  Interator it = p.getHaLavorato();
  while (it.hasNext()) {
    Azienda az = (Azienda)it.next();
    System.out.println(az.getRagioneSociale());
  }
}
```


Quinto caso: associazioni 0..* con attributi

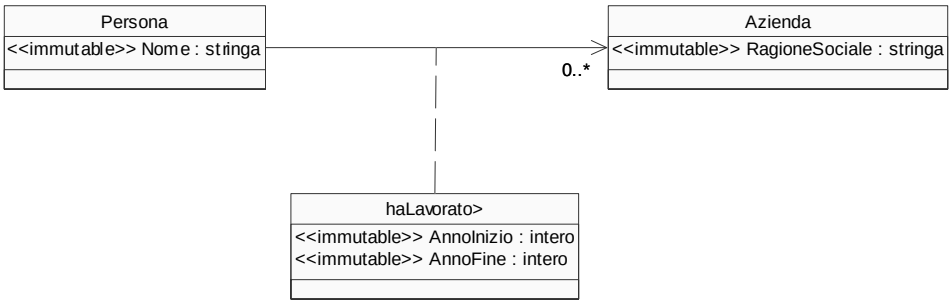
Ci concentriamo ora su associazioni binarie con molteplicità 0..*, e **con attributi**. Ci riferiremo al seguente esempio.

Schema concettuale da realizzare in Java (solo la classe *Persona* ha responsabilità sull'associazione):



Assumiamo per semplicità che si lavori sempre per anni interi.

Diagramma Realizzativo: Quinto Caso



Quinto caso (cont.)

Dobbiamo combinare le scelte fatte in precedenza:

1. come per tutte le associazioni con attributi, dobbiamo definire una apposita classe Java per la rappresentazione del link (`TipoLinkHaLavorato`);

2. come per tutte le associazioni con vincolo di molteplicità 0..*, dobbiamo utilizzare una struttura di dati per la rappresentazione dei link.

Quinto caso: rappresentazione dei link

La classe Java `TipoLinkHaLavorato` per la rappresentazione dei link deve gestire:

- gli attributi dell'associazione (*AnnoInizio*, *AnnoFine*);
- i riferimenti agli oggetti relativi al link (di classe `Persona` e `Azienda`).

Pertanto, avrà gli opportuni campi dati e funzioni (costruttori e `get`). Inoltre, avrà la funzione `equals` per verificare l'uguaglianza solo sugli oggetti collegati dal link, ignorando gli attributi.

Rappresentazione dei link in Java

```
// File ParteQuarta/Ass01STARAttr/TipoLinkHaLavorato

public class TipoLinkHaLavorato {
  private final Persona laPersona;
  private final Azienda laAzienda;
  private final int annoInizio, annoFine;
  public TipoLinkHaLavorato(Azienda x, Persona y, int ai, int af) {
    laAzienda = x; laPersona = y; annoInizio = ai; annoFine = af;
  }
  public Azienda getAzienda() { return laAzienda; }
  public Persona getPersona() { return laPersona; }
  public int getAnnoInizio() { return annoInizio; }
  public int getAnnoFine() { return annoFine; }

  public boolean equals(Object o) {
    if (o != null && getClass().equals(o.getClass())) {
```

```
      TipoLinkHaLavorato b = (TipoLinkHaLavorato)o;
      return b.laPersona != null && b.laAzienda != null &&
             b.laPersona == laPersona && b.laAzienda == laAzienda;
    }
    else return false;
  }
}
```

Esempio del quinto caso: la classe Persona

La classe Java `Persona` avrà un campo per la rappresentazione di tutti i link relativi ad un oggetto della classe.

Scegliamo ancora di utilizzare l'interfaccia Java `Set` realizzata dalla classe `InsiemeArrayOmogeneo`.

La funzione `inserisciLinkHaLavorato()` deve effettuare tutti i controlli necessari per mantenere la consistenza dei riferimenti (già visti per il caso 0..1).

Analogamente, la funzione `eliminaLinkHaLavorato()` deve assicurarsi che:

- l'oggetto che rappresenta il link esista;
- che il link da eliminare sia quello effettivamente contenuto nel insieme di link dell'oggetto di invocazione.

La classe Java Persona

```
// File ParteQuarta/AssOSTARAttr/Persona.java
import java.util.*;
import insiemearray.*;

public class Persona {
  private final String nome;
  private Set insieme_link;
  public Persona(String n) {
    nome = n;
    insieme_link = new InsiemeArrayOmogeneo(TipoLinkHaLavorato.class);
  }
  public String getNome() { return nome; }
  public void inserisciLinkHaLavorato(TipoLinkHaLavorato t) {
    if (t != null && t.getAzienda() != null && t.getPersona() == this)
      insieme_link.add(t);
  }
```



```
  public void eliminaLinkHaLavorato(TipoLinkHaLavorato t) {
    if (t != null)
      insieme_link.remove(t);
  }
  public Iterator getLinkHaLavorato() {
    return insieme_link.iterator();
  }
}
```

Esempio di cliente

La seguente funzione stampa le ragioni sociali, l'anno di inizio e di fine rapporto di tutte le aziende per cui una certa persona ha lavorato.

```
public static void stampaAziende(Persona p) {
  System.out.println(p.getNome() +
                     " ha lavorato nelle seguenti aziende:");
  Iterator it = p.getLinkHaLavorato();
  while (it.hasNext()) {
    TipoLinkHaLavorato lnk = (TipoLinkHaLavorato)it.next();
    System.out.println(lnk.getAzienda().getRagioneSociale() + " dall'anno "
                       + lnk.getAnnoInizio() + " all'anno " +
                       lnk.getAnnoFine());
  }
}
```


Esercizio 5: cliente

Realizzare in Java lo use case *Analisi Mercato Lavoro*:

InizioSpecificaUseCase Analisi Mercato Lavoro

PeriodoPiùLungo *(p: Persona): intero*
pre: nessuna
post: *result* è il periodo consecutivo (in anni) più lungo in cui *p* ha lavorato per la stessa azienda

riAssuntoSubito *(p: Persona): booleano*
pre: nessuna
post: *result* vale *true* se e solo se *p* ha lavorato consecutivamente per due aziende (anno di inizio per un'azienda uguale all'anno di fine per un'altra azienda + 1)

SonoStatiColleghi *(p1: Persona, p2: Persona): booleano*
pre: nessuna
post: *result* vale *true* se e solo se *p1* e *p2* hanno lavorato contemporaneamente per la stessa azienda

FineSpecifica

Soluzione esercizio 5

```
// File ParteQuarta/AssOSTARAttr/AnalisiMercatoLavoro.java
import java.util.*;

public class AnalisiMercatoLavoro {
  public static int periodoPiuLungo(Persona p) {
    // restituisce il periodo consecutivo (in anni) più lungo
    // in cui la persona p ha lavorato per la stessa azienda
    int max = 0;
    Iterator it = p.getLinkHaLavorato();
    while (it.hasNext()) {
      TipoLinkHaLavorato lnk = (TipoLinkHaLavorato)it.next();
      int durata = lnk.getAnnoFine() - lnk.getAnnoInizio() + 1;
      if (durata > max)
        max = durata;
    }
    return max;
  }
  public static boolean riAssuntoSubito(Persona p) {
    // restituisce true se e solo se la persona p ha lavorato
    // consecutivamente per due aziende (anno di inizio per un'azienda uguale
    // all'anno di fine per un'altra azienda + 1)
    Iterator it = p.getLinkHaLavorato();
    // le aziende per cui p ha lavorato
    while (it.hasNext()) {
```



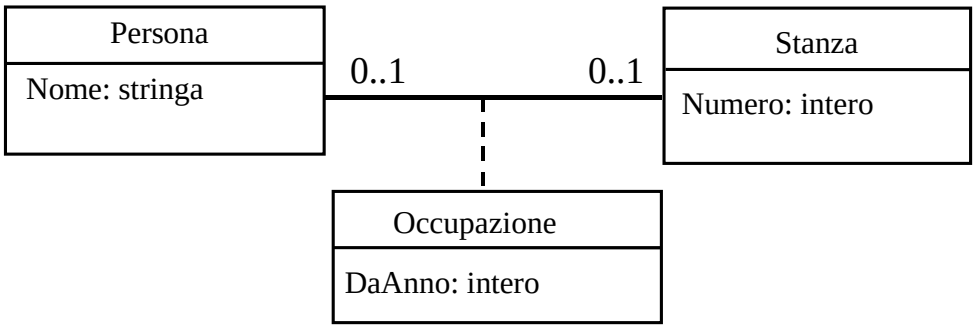
```
      TipoLinkHaLavorato lnk = (TipoLinkHaLavorato)it.next();
      Iterator it2 = p.getLinkHaLavorato();
      while (it2.hasNext()) {
        TipoLinkHaLavorato lnk2 = (TipoLinkHaLavorato)it2.next();
        if (lnk.getAnnoFine() == lnk2.getAnnoInizio() - 1)
          return true;
      }
    }
    return false;
  }
  public static boolean sonoStatiColleghi(Persona p1, Persona p2) {
    // restituisce true se e solo se p1 e p2 hanno lavorato
    // contemporaneamente per la stessa azienda
    Iterator it = p1.getLinkHaLavorato();
    while (it.hasNext()) {
      TipoLinkHaLavorato lnk = (TipoLinkHaLavorato)it.next();
      if (appartiene(lnk,p2.getLinkHaLavorato()))
        return true;
    }
    return false;
  }
  private static boolean appartiene(TipoLinkHaLavorato t, Iterator it) {
    // funzione di servizio: verifica se nell'insieme di link ins
    // sia presente un link compatibile con t
    while (it.hasNext()) {
      TipoLinkHaLavorato lnk = (TipoLinkHaLavorato)it.next();
```

```
      if (compatibili(t,lnk))
        return true;
    }
    return false;
  }
  private static boolean compatibili(TipoLinkHaLavorato t1,
                                     TipoLinkHaLavorato t2) {
    // funzione di servizio: verifica se i link t1 e t2 sono "compatibili",
    // ovvero se si riferiscono alla stessa azienda e a periodi temporali
    // con intersezione non nulla
    return t1.getAzienda() == t2.getAzienda() && // UGUAGLIANZA SUPERFICIALE
      t2.getAnnoFine() >= t1.getAnnoInizio() &&
      t2.getAnnoInizio() <= t1.getAnnoFine();
  }
  public static void analisi(Persona p) {
    System.out.println("Il periodo piu' lungo in cui " + p.getNome() +
                       " ha lavorato per la stessa azienda e' di anni: " +
                       periodoPiuLungo(p));
    System.out.println(p.getNome() + (riAssuntoSubito(p)?" ":" non ") +
                       "ha lavorato consecutivamente per due aziende");
  }
}
```

Sesto caso: responsabilità di entrambe le classi

Affrontiamo il caso di associazione binaria in cui **entrambe le classi abbiano la responsabilità sull'associazione**. Per il momento, assumiamo che la molteplicità sia 0..1 per entrambe le classi.

Ci riferiremo al seguente esempio, assumendo che sia *Persona* sia *Stanza* abbiano responsabilità sull'associazione.



Resp. di entrambe le classi (cont.)

Problema di fondo:
quando creiamo un link fra un oggetto Java `pe` di classe `Persona` ed un oggetto Java `st` di classe `Stanza`, dobbiamo cambiare lo stato **sia di** `pe` **sia di** `st`.

In particolare:

- l'oggetto `pe` si deve riferire all'oggetto `st`;
- l'oggetto `st` si deve riferire all'oggetto `pe`.

Discorso analogo vale quando **eliminiamo** un link fra due oggetti.

Resp. di entrambe le classi (cont.)

Chiaramente, non possiamo dare al cliente delle classi `Persona` e `Stanza` questo onere, che deve essere gestito invece dai moduli server.

Una possibile soluzione è di assegnare questo onere ad entrambe le classi. Ma questo implicherebbe che entrambe le classi dovrebbero gestire una struttura di dati per la stessa associazione, con duplicazione di risorse di spazio di memoria e di tempo per la gestione. Per ragioni di efficienza, è preferibile quindi **centralizzare** la responsabilità di assegnare i riferimenti in maniera corretta.

In particolare, realizziamo una ulteriore classe Java (chiamata `Associazione-Occupazione`) che gestisce la corretta creazione della rete dei riferimenti. Questa classe è di fatto un modulo per l'inserimento e la cancellazione di link di tipo *Occupazione*. Ogni suo oggetto ha un riferimento ad un oggetto Java che rappresenta un link di tipo *Occupazione*.

Continuiamo ad usare una classe per i link, in questo caso la classe Java `TipoLinkOccupazione`, che modella tuple del prodotto cartesiano tra *Stanza* e *Persona* con attributo *DaAnno*.

Caratteristiche delle classi Java

`Persona`: oltre ai campi dati e funzione per la gestione dei suoi attributi, avrà:

- un campo di tipo `TipoLinkOccupazione`, inizializzato a `null`;
- funzioni per la gestione di questo campo, in particolare:
  - `void inserisciLinkOccupazione(AssociazioneOccupazione)`, per associare un link all'oggetto;
  - `void eliminaLinkOccupazione(AssociazioneOccupazione)`, per rimuovere l'associazione di un link all'oggetto;
  - `TipoLinkOccupazione getLinkOccupazione()`, per interrogare l'oggetto riguardo all'eventuale link.

`Stanza`: del tutto simile a `Persona`.

La classe Java Persona

```
// File ParteQuarta/RespEntrambi01/Persona.java
public class Persona implements SupportoAssociazioneOccupazione {

  private final String nome;
  private TipoLinkOccupazione link;

  public Persona(String n) { nome = n; }
  public String getNome() { return nome; }


  public TipoLinkOccupazione getLinkOccupazione() {
      return link;
  }

  public void inserisciLinkOccupazione(AssociazioneOccupazione a) {
    if (a != null && a.getLink().getPersona() == this &&
```

```
        link == null)  //nota molt. 0..1: posso inserire se non c'e' gia'
                       //un link
      link = a.getLink();
    else throw new RuntimeException("Inserimento link impossibile!");
  }

  public void eliminaLinkOccupazione(AssociazioneOccupazione a) {
    if (a != null &&
        a.getLink().equals(link))
      link = null;
    else throw new RuntimeException("Eliminazione link impossibile!");
  }
}
```

La classe Java Stanza

```
// File ParteQuarta/RespEntrambi01/Stanza.java
public class Stanza implements SupportoAssociazioneOccupazione {

  private final int numero;
  private TipoLinkOccupazione link;
  public Stanza(int n) { numero = n; }

  public int getNumero() { return numero; }
  public TipoLinkOccupazione getLinkOccupazione() {
      return link;
  }

  public void inserisciLinkOccupazione(AssociazioneOccupazione a) {
    if ( a != null && a.getLink().getStanza() == this &&
         link == null)
      link = a.getLink();
```

```
    else throw new RuntimeException("Inserimento link impossibile!");
  }
  public void eliminaLinkOccupazione(AssociazioneOccupazione a) {
    if (a != null &&
        a.getLink().equals(link))
      link = null;
    else throw new RuntimeException("Eliminazione link impossibile!");
  }
}
```

Caratteristiche delle classi Java (cont.)

`TipoLinkOccupazione`: sarà del tutto simile al caso in cui la responsabilità sull'associazione è singola. Avrà:

- tre campi dati (per la stanza, per la persona e per l'attributo dell'associazione);
- un costruttore, che inizializza questi campi utilizzando i suoi argomenti;
- tre funzioni `get`, per interrogare l'oggetto;
- la funzione `equals` per verificare l'uguaglianza solo sugli oggetti collegati dal link, ignorando gli attributi.

La classe Java TipoLinkOccupazione

```
// File ParteQuarta/RespEntrambi01/TipoLinkOccupazione.java

public class TipoLinkOccupazione {
  private final Stanza laStanza;
  private final Persona laPersona;
  private final int daAnno;
  public TipoLinkOccupazione(Stanza x, Persona y, int a) {
    laStanza = x; laPersona = y; daAnno = a;
  }
  public Stanza getStanza() { return laStanza; }
  public Persona getPersona() { return laPersona; }
  public int getDaAnno() { return daAnno; }

  public boolean equals(Object o) {
    if (o != null && getClass().equals(o.getClass())) {
      TipoLinkOccupazione b = (TipoLinkOccupazione)o;
```



```
      return b.laPersona != null && b.laStanza != null &&
             b.laPersona == laPersona && b.laStanza == laStanza;
    }
    else return false;
  }
}
```

Caratteristiche delle classi Java (cont.)

`AssociazioneOccupazione`: avrà:

- un campo dato, di tipo `TipoLinkOccupazione` per la rappresentazione del link;
- funzioni per la gestione di questo campo, in particolare:
  - `static void inserisci(TipoLinkOccupazione)`, per stabilire un link fra una persona ed una stanza;
  - `static void elimina(TipoLinkOccupazione)`, per rimuovere un link fra una persona ed una stanza;
  - `TipoLinkOccupazione getLink()`, per ottenere il link;
- il costruttore sarà **privato**.

La classe Java AssociazioneOccupazione

```
// File ParteQuarta/RespEntrambi01/AssociazioneOccupazione.java

public class AssociazioneOccupazione {

  private TipoLinkOccupazione link;
  private AssociazioneOccupazione(TipoLinkOccupazione x) { link = x; }
  public TipoLinkOccupazione getLink() { return link; }

  public static void inserisci(TipoLinkOccupazione y) {
      if (y != null && y.getPersona() != null && y.getStanza() != null) {
            AssociazioneOccupazione k = new AssociazioneOccupazione(y);
            y.getStanza().inserisciLinkOccupazione(k);
            y.getPersona().inserisciLinkOccupazione(k);
      }
  }
  public static void elimina(TipoLinkOccupazione y) {
```

```
        if (y != null && y.getPersona() != null && y.getStanza() != null) {
                AssociazioneOccupazione k = new AssociazioneOccupazione(y);
                y.getStanza().eliminaLinkOccupazione(k);
                y.getPersona().eliminaLinkOccupazione(k);
        }
    }
}
```

La classe Java AssociazioneOccupazione (cont.)

Il costruttore della classe `AssociazioneOccupazione` è **privato** in quanto **non vogliamo che i clienti siano in grado di creare oggetti di questa classe**.

I clienti saranno in grado di:

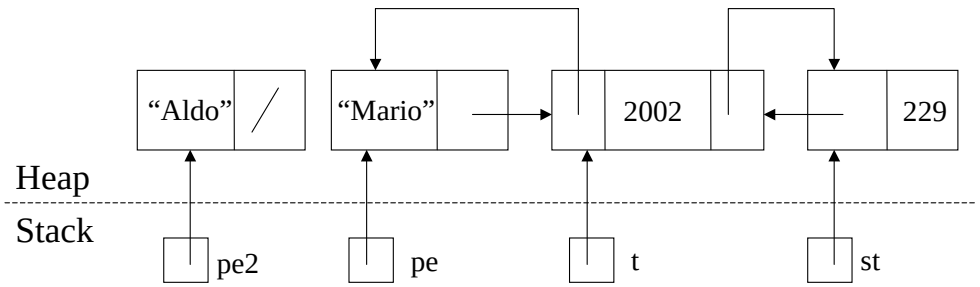
- creare link, di tipo `TipoLinkOccupazione`, stabilendo contestualmente la stanza, la persona e l'anno;
- associare link agli oggetti di classe `Stanza` e `Persona`, mediante una chiamata alla funzione `AssociazioneOccupazione.inserisci()`;
- rimuovere link, mediante una chiamata alla funzione `AssociazioneOccupazione.elimina()`.

Considerazioni sulle classi Java

- Le funzioni `inserisciLinkOccupazione()` ed `eliminaLinkOccupazione()` della classe `Persona` di fatto possono essere invocate **solamente dalla classe** `AssociazioneOccupazione`, in quanto:
  - per invocarle dobbiamo passare loro degli argomenti di tipo `AssociazioneOccupazione`, e
  - gli oggetti della classe `AssociazioneOccupazione` non possono essere creati, se non attraverso le funzioni `inserisci()` ed `elimina()` di quest'ultima.

- Forziamo quindi i clienti che vogliono stabilire o rimuovere dei link ad usare le funzioni (statiche, in quanto svincolate da oggetti di invocazione) `inserisci()` ed `elimina()` della classe `AssociazioneOccupazione`, e non effettuare inserimenti e cancellazioni di link direttamente mediante le classi `Persona` e `Stanza`.

Possibile stato della memoria



Due oggetti di classe `Persona`, di cui uno con una stanza associata ed uno no.

Si noti che l'oggetto di classe `AssociazioneOccupazione` non è direttamente accessibile dai clienti.

Realizzazione della situazione di esempio

```
public static void main (String[] args) {

  Stanza st = new Stanza(229);

  Persona pe = new Persona("Mario");
  Persona pe2 = new Persona("Aldo");

  TipoLinkOccupazione t = new TipoLinkOccupazione(st,pe,2002);

  AssociazioneOccupazione.inserisci(t);
}
```


Esercizio 6: cliente

Realizzare in Java lo use case *Riallocazione Personale*:

InizioSpecificaUseCase Riallocazione Personale

Promuovi *(ins: Insieme(Persona), st: Stanza, anno: intero)*
pre: *ins* non è vuoto; a tutte le persone di *ins* è assegnata una stanza
post: alla persona di *ins* che è da più tempo nella stessa stanza viene assegnata la stanza *st*, a partire dall'anno *anno*

Libera *(ins: Insieme(Stanza))*
pre: a tutte le persone in *ins* è assegnata una stanza
post: le stanze di *ins* che sono occupate da più tempo vengono liberate

...

Esercizio 6: cliente (cont.)

...

Trasloca *(ins1: Insieme(Persona), ins2: Insieme(Persona), anno: intero)*

pre: *ins1* e *ins2* hanno la stessa cardinalità; a tutte le persone di *ins2* è assegnata una stanza

post: ad ogni persona di *ins1* viene assegnata una stanza di una persona di *ins2*, togliendola a quest'ultima, a partire dall'anno *anno*

FineSpecifica

Soluzione esercizio 6

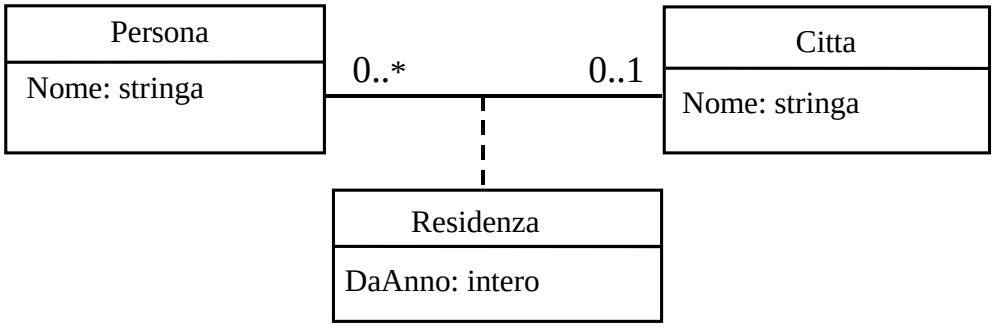
```
// File ParteQuarta/RespEntrambi01/Main.java
import java.util.*;
import insiemearray.*;

public class RiallocazionePersonale {
  public static void promuovi(Set ins, Stanza st, int anno) {
    // alla persona di ins che è da più tempo nella
    // stessa stanza viene assegnata la stanza st,
    // a partire dall'anno passato come argomento
    Iterator it = ins.iterator();
    int min = ((Persona)it.next()).getLinkOccupazione().getDaAnno();
    while (it.hasNext()) {
      Persona p = (Persona)it.next();
      if (p.getLinkOccupazione().getDaAnno() < min)
        min = p.getLinkOccupazione().getDaAnno();
    }
    it = ins.iterator();
    boolean trovato = false;
    while (it.hasNext()) {
      Persona p = (Persona)it.next();
      if (p.getLinkOccupazione().getDaAnno() == min) {
        AssociazioneOccupazione.elimina(p.getLinkOccupazione());
        TipoLinkOccupazione t = new TipoLinkOccupazione(st,p,anno);
        AssociazioneOccupazione.inserisci(t);
```

```
        return;
      }
    }
  }
  public static void libera(Set ins) {
    // tutte le stanze di ins che sono impegnate da più tempo vengono liberate
    if (ins.isEmpty()) return;
    Iterator it = ins.iterator();
    int min = ((Stanza)it.next()).getLinkOccupazione().getDaAnno();
    while (it.hasNext()) {
      Stanza s = (Stanza)it.next();
      if (s.getLinkOccupazione().getDaAnno() < min)
        min = s.getLinkOccupazione().getDaAnno();
    }
    it  = ins.iterator();
    while (it.hasNext()) {
      Stanza s = (Stanza)it.next();
      if (s.getLinkOccupazione().getDaAnno() == min)
        AssociazioneOccupazione.elimina(s.getLinkOccupazione());
    }
  }
  public static void trasloca(Set ins1, Set ins2, int anno) {
    // ins1 e ins2 sono insiemi di Persona della stessa cardinalità;
    // ad ogni persona di ins1 viene assegnata una stanza di una persona
    // di ins2, togliendola a quest'ultima, a partire dall'anno passato
    // come argomento
```

```
    Iterator it1 = ins1.iterator();
    Iterator it2 = ins2.iterator();
    while (it1.hasNext()) {
      Persona p1 = (Persona)it1.next();
      Persona p2 = (Persona)it2.next();
      TipoLinkOccupazione t2 = p2.getLinkOccupazione();
      Stanza st = t2.getStanza();
      TipoLinkOccupazione t1 = new TipoLinkOccupazione(st,p1,anno);
      AssociazioneOccupazione.elimina(t2);
      AssociazioneOccupazione.inserisci(t1);
    }
  }
}
```

Settimo caso: resp. di due classi, molt. 0..*

Affrontiamo il caso di associazione binaria in cui **entrambe le classi abbiano la responsabilità sull'associazione, ed in cui una delle molteplicità sia 0..***. Ci riferiremo al seguente esempio.



Supponiamo che sia *Persona* sia *Città* abbiano responsabilità sull'associazione. Per semplificare, ammettiamo che una persona possa non risiedere in alcuna città (vincolo di molteplicità 0..1).

Resp. di entrambe le classi: molt. 0..*

La metodologia proposta per la molteplicità 0..1 può essere usata anche per la molteplicità 0..* (per il momento, una delle due molteplicità è ancora 0..1). Le differenze principali sono:

- La classe Java (nel nostro esempio: `Citta`) i cui oggetti possono essere legati a più oggetti dell'altra classe Java (nel nostro esempio: `Persona`) ha le seguenti caratteristiche:
  - ha un ulteriore campo dato di tipo `Set`, per poter rappresentare tutti i link; l'oggetto di classe `InsiemeArrayOmogeneo` che implementa `Set` viene creato tramite il costruttore;
  - ha un campo funzione pubblico `getLinkResidenza()` che permette di ottenere tutti i link residenza di una persona, enumerandoli attraverso un iteratore.

– infine implementa l'interfaccia `SopportoAssociazioneResidenza` realizzando le funzioni (`inserisciLinkResidenza()`, `eliminaLinkResidenza()` per la gestione dell'insieme dei link.

La classe Java Citta

```
// File ParteQuarta/RespEntrambiOSTAR/Citta.java
import java.util.*;
import insiemearray.*;

public class Citta implements SupportoAssociazioneResidenza {
  private final String nome;
  private Set insieme_link;

  public Citta(String n) {
    nome = n;
    insieme_link = new InsiemeArrayOmogeneo(TipoLinkResidenza.class);
  }

  public String getNome() { return nome; }

  public Iterator getLinkResidenza() {
```

```
    return insieme_link.iterator();
  }

  public void inserisciLinkResidenza(AssociazioneResidenza a) {
    if (a != null  &&
        a.getLink().getCitta() == this)
      insieme_link.add(a.getLink());
    else throw new RuntimeException("Inserimento link impossibile!");
  }
  public void eliminaLinkResidenza(AssociazioneResidenza a) {
    if (a != null)
      insieme_link.remove(a.getLink());
    else throw new RuntimeException("Eliminazione link impossibile!");
  }
}
```

Resp. di entrambe le classi: molt. 0..* (cont.)

- La classe Java (nel nostro esempio: `Persona`) i cui oggetti possono essere legati ad un singolo oggetto dell'altra classe Java (nel nostro esempio: `Citta`) è **esattamente identica** al caso della molteplicità 0..1.

- Analogamente, la classe Java per la rappresentazione dei link per la rappresentazione di tuple del prodotto cartesiano tra *Città* e *Persona* con attributo *DaAnno* (nel nostro esempio `TipoLinkResidenza`) è **esattamente identica** al caso della molteplicità 0..1.

La classe Java Persona

```
// File ParteQuarta/RespEntrambiOSTAR/Persona.java

public class Persona implements SupportoAssociazioneResidenza {
  private final String nome;
  private TipoLinkResidenza link;
  public Persona(String n) { nome = n; }
  public String getNome() { return nome; }
  public TipoLinkResidenza getLinkResidenza() {
    return link;
  }
  public void inserisciLinkResidenza(AssociazioneResidenza a) {
    if (a != null && a.getLink().getPersona() == this &&
        link == null)
      link = a.getLink();
    else throw new RuntimeException("Inserimento link impossibile!");
  }
```



```
  public void eliminaLinkResidenza(AssociazioneResidenza a) {
    if (a != null&&
        a.getLink().equals(link))
      link = null;
    else throw new RuntimeException("Eliminazione link impossibile!");
  }
}
```

La classe Java TipoLinkOccupazione

```
// File ParteQuarta/RespEntrambiOSTAR/TipoLinkResidenza.java

public class TipoLinkResidenza {
  private final Citta laCitta;
  private final Persona laPersona;
  private final int daAnno;
  public TipoLinkResidenza(Citta x, Persona y, int a) {
    laCitta = x; laPersona = y; daAnno = a;
  }
  public Citta getCitta() { return laCitta; }
  public Persona getPersona() { return laPersona; }
  public int getDaAnno() { return daAnno; }
  public boolean equals(Object o) {
    if (o != null && getClass().equals(o.getClass())) {
      TipoLinkResidenza b = (TipoLinkResidenza)o;
      return b.laPersona != null && b.laCitta != null &&
```



```
             b.laPersona == laPersona && b.laCitta == laCitta;
    }
    else return false;
  }
}
```

La classe Java AssociazioneResidenza

```
// File ParteQuarta/RespEntrambi01/AssociazioneResidenza.java

public class AssociazioneResidenza {
  private AssociazioneResidenza(TipoLinkResidenza x) { link = x; }
  private TipoLinkResidenza link;
  public TipoLinkResidenza getLink() { return link; }
  public static void inserisci(TipoLinkResidenza y) {
    if (y != null && y.getPersona() != null && y.getCitta() != null) {
          AssociazioneResidenza k = new AssociazioneResidenza(y);
          y.getCitta().inserisciLinkResidenza(k);
          y.getPersona().inserisciLinkResidenza(k);
    }
  }
  public static void elimina(TipoLinkResidenza y) {
    if (y != null && y.getPersona() != null && y.getCitta() != null) {
          AssociazioneResidenza k = new AssociazioneResidenza(y);
```



```
          y.getCitta().eliminaLinkResidenza(k);
          y.getPersona().eliminaLinkResidenza(k);
    }
  }
}
```

Esercizio 7: cliente

Realizzare in Java lo use case *Gestione Anagrafe*:

InizioSpecificaUseCase Gestione Anagrafe

TrovaNuovi *(c: Città, anno: intero): Insieme(Persona)*
pre: nessuna
post: *result* è l'insieme di persone che sono residenti nella città *c* da non prima dell'anno *a*

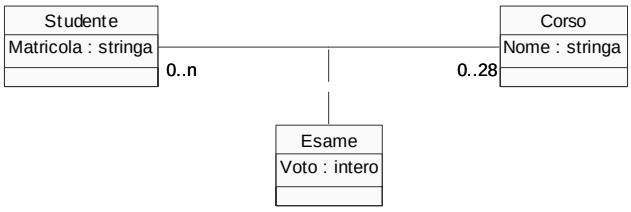
FineSpecifica

Soluzione esercizio 7

```
// File ParteQuarta/RespEntrambiOSTAR/GestioneAnagrafe.java
import java.util.*;
import insiemearray.*;

public class GestioneAnagrafe {
  public static Set trovaNuovi(Citta c, int anno) {
    Iterator it = c.getLinkResidenza();
    Set out = new InsiemeArrayOmogeneo(Persona.class);
    while (it.hasNext()) {
      TipoLinkResidenza lnk = (TipoLinkResidenza)it.next();
      if (lnk.getDaAnno() >= anno)
          out.add(lnk.getPersona());
    }
    return out;
  }
}
```

Ottavo caso: resp. di due classi, molt. 0..*, 0..*

Affrontiamo il caso di associazione binaria in cui **entrambe le classi abbiano la responsabilità sull'associazione, ed entrambe con molteplicità 0..***. Ci riferiremo al seguente esempio.



Supponiamo che sia *Studente* sia *Corso* abbiano responsabilità sull'associazione.

Entrambe le molteplicità sono 0..* (cont.)

La stessa metodologia proposta per il caso in cui entrambe le classi abbiano responsabilità sull'associazione può essere usata anche quando entrambe le molteplicità sono 0..*.

In particolare, ora le due classi Java sono strutturalmente simili:

- hanno un ulteriore campo dato di tipo `Set`, per poter rappresentare tutti i link;

  l'oggetto di classe `InsiemeArrayOmogeneo` che implementa `Set` viene creato tramite il costruttore;

- ha un campo funzione pubblico `getLinkEsame()` che permette di ottenere tutti i link esame, enumerandoli attraverso un iteratore.

- infine implementa l'interfaccia `SopportoAssociazioneEsame` realizzando le funzioni (`inserisciLinkEsame()`, `eliminaLinkEsame()` per la gestione dell'insieme dei link.

Riportiamo il codice di tutte le classi.

La classe Java Studente

```
// File ParteQuarta/RespEntrambiOSTAR2/Studente.java
import java.util.*;
import insiemearray.*;

public class Studente  implements SupportoAssociazioneEsame {
  private final String matricola;
  private Set insieme_link;
  public Studente(String n) {
    matricola = n;
    insieme_link = new InsiemeArrayOmogeneo(TipoLinkEsame.class);
  }
  public String getMatricola() { return matricola; }

  public Iterator getLinkEsame() {
    return insieme_link.iterator();
  }
```

```
  public void inserisciLinkEsame(AssociazioneEsame a) {
    if (a != null  && a.getLink().getStudente() == this)
      insieme_link.add(a.getLink());
    else throw new RuntimeException("Inserimento link impossibile!");
  }
  public void eliminaLinkEsame(AssociazioneEsame a) {
    if (a != null)
      insieme_link.remove(a.getLink());
    else throw new RuntimeException("Eliminazione link impossibile!");
  }
}
```

La classe Java Corso

```
// File ParteQuarta/RespEntrambi0STAR2/Corso.java
import java.util.*;
import insiemearray.*;

public class Corso implements SupportoAssociazioneEsame {
  private final String nome;
  private Set insieme_link;
  public Corso(String n) {
    nome = n;
    insieme_link = new InsiemeArrayOmogeneo(TipoLinkEsame.class);
  }
  public String getNome() { return nome; }

  public Iterator getLinkEsame() {
    return insieme_link.iterator();
  }
```

```
  public void inserisciLinkEsame(AssociazioneEsame a) {
    if (a != null && a.getLink().getCorso() == this)
      insieme_link.add(a.getLink());
    else throw new RuntimeException("Inserimento link impossibile!");
  }
  public void eliminaLinkEsame(AssociazioneEsame a) {
    if (a != null)
      insieme_link.remove(a.getLink());
    else throw new RuntimeException("Eliminazione link impossibile!");
  }
}
```

La classe Java TipoLinkEsame

```
// File ParteQuarta/RespEntrambiOSTAR2/TipoLinkEsame.java

public class TipoLinkEsame {
  private final Corso ilCorso;
  private final Studente loStudente;
  private final int voto;
  public TipoLinkEsame(Corso x, Studente y, int a) {
    ilCorso = x; loStudente = y; voto = a;
  }
  public Corso getCorso() { return ilCorso; }
  public Studente getStudente() { return loStudente; }
  public int getVoto() { return voto; }
  public boolean equals(Object o) {
    if (o != null && getClass().equals(o.getClass())) {
      TipoLinkEsame b = (TipoLinkEsame)o;
      return b.ilCorso != null && b.loStudente != null &&
```

```
            b.ilCorso == ilCorso && b.loStudente == loStudente;
    }
    else return false;
  }
}
```

La classe Java AssociazioneEsame

```
// File ParteQuarta/RespEntrambiOSTAR2/AssociazioneEsame.java

public class AssociazioneEsame {
  private AssociazioneEsame(TipoLinkEsame x) { link = x; }
  private TipoLinkEsame link;
  public TipoLinkEsame getLink() { return link; }
  public static void inserisci(TipoLinkEsame y) {
    if (y != null && y.getStudente() != null && y.getCorso() != null) {
      AssociazioneEsame k = new AssociazioneEsame(y);
      k.link.getCorso().inserisciLinkEsame(k);
      k.link.getStudente().inserisciLinkEsame(k);
    }
  }
  public static void elimina(TipoLinkEsame y) {
    if (y != null && y.getStudente() != null && y.getCorso() != null) {
      AssociazioneEsame k = new AssociazioneEsame(y);
```

```
      k.link.getCorso().eliminaLinkEsame(k);
      k.link.getStudente().eliminaLinkEsame(k);
    }
  }
}
```

Esercizio 8: cliente

Realizzare in Java lo use case *Valutazione Didattica*:

InizioSpecificaUseCase Valutazione Didattica

StudenteBravo *(s: Studente): booleano*

pre: nessuna

post: *result* è *true* se e solo se tutti gli esami sostenuti dallo studente *s* sono stati superati con voto non inferiore a 27

CorsoFacile *(c: Corso): booleano*

pre: nessuna

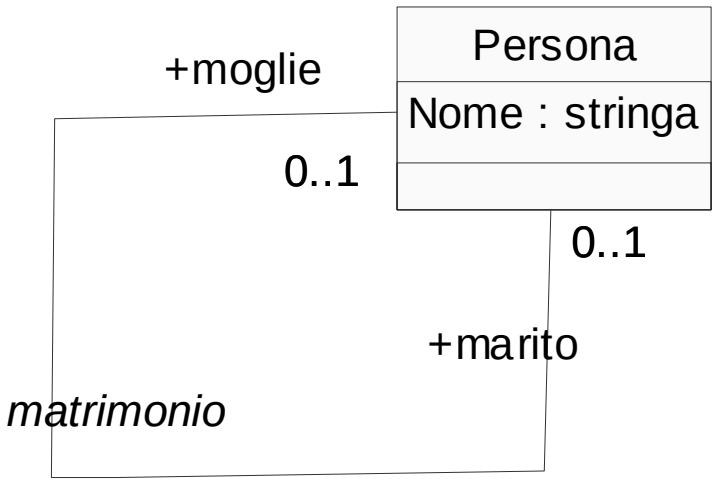
post: *result* è *true* se e solo se tutti gli esami sostenuti per il corso *c* sono stati superati con voto non inferiore a 27

FineSpecifica

Soluzione esercizio 8

```
// File ParteQuarta/RespEntrambi0STAR2/ValutazioneDidattica.java
import java.util.*;

public class ValutazioneDidattica {
  public static boolean studenteBravo(Studente s) {
    Iterator it = s.getLinkEsame();
    while (it.hasNext()) {
      TipoLinkEsame lnk = (TipoLinkEsame)it.next();
      if (lnk.getVoto() < 27)
          return false;
    }
    return true;
  }
  public static boolean corsoFacile(Corso c) {
    Iterator it = c.getLinkEsame();
    while (it.hasNext()) {
      TipoLinkEsame lnk = (TipoLinkEsame)it.next();
      if (lnk.getVoto() < 27)
          return false;
    }
    return true;
  }
}
```


Caso Particolare



Supponiamo che `Persona` abbia responsabilità su matrimonio in entrambi i ruoli, di marito e di moglie.

Caso Particolare (cont.)

In questo caso, sono necessarie due interfacce di supporto alla classe `Associazione`:

```
public interface SupportoAssociazioneMatrimonioconRuoloMarito {
  void inserisciLinkMatrimonioconRuoloMarito(AssociazioneMatrimonio a);
  void eliminaLinkMatrimonioconRuoloMarito(AssociazioneMatrimonio a);
}

public interface SupportoAssociazioneMatrimonioconRuoloMoglie {
  void inserisciLinkMatrimonioconRuoloMoglie(AssociazioneMatrimonio a);
  void eliminaLinkMatrimonioconRuoloMoglie(AssociazioneMatrimonio a);
}
```

- La classe `Persona` deve implementare entrambe le interfacce.
- Le funzioni `inserisci` ed `elimina` in `AssocizioneMatrimonio` faranno uso di entrambe le interfacce.

Altre molteplicità di associazione

Per quanto riguarda le altre molteplicità di associazione, tratteremo i seguenti due casi:

1. molteplicità massima finita (cioè diversa da *),
2. molteplicità minima diversa da zero.

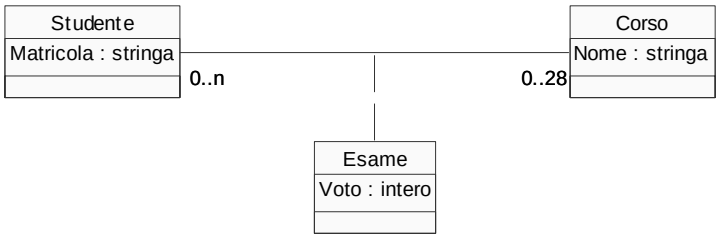
In generale, prevediamo che la classe rispetto a cui esiste uno dei vincoli di cui sopra debba essere in grado di **verificare le moteplicità** che la riguardano. Ne segue che tale classe **ha necessariamente responsabilità sull'associazione**.

Altre molteplicità di associazione (cont.)

L'ideale sarebbe fare in modo che tutti i vincoli di molteplicità di un diagramma delle classi fossero rispettati *in ogni momento* nella realizzazione in Java. Ma ciò è, in generale, molto complicato.

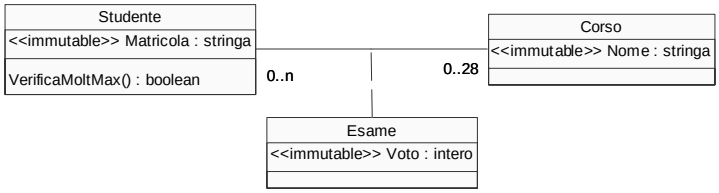
La strategia che seguiremo semplifica il problema, **ammettendo che gli oggetti possano essere in uno stato che non rispetta vincoli di molteplicità massima o vincoli di molteplicità minima, ma lanciando una eccezione** nel momento in cui un cliente chieda di utilizzare un link (relativo ad una associazione `A`) di un oggetto che non rispetta tali vincoli sull'associazione `A`.

Nono caso: molteplicità massima diversa da *



Studente ha responsabilità sull'associazione perchè ha molteplicità massima diversa da *. Supponiamo che anche *Corso* abbia responsabilità sull'associazione.

Diagramma Realizzativo Nono Caso



La classe *Studente* ha una operazioni aggiuntiva *VerificaMoltMax()* che consente di effettuare una verifica sulla cardinalità massima degli oggetti della classe *Studente* nell'associazione *Esame*.

Molteplicità massima diversa da * (cont.)

Rispetto al caso in cui il vincolo di molteplicità sia 0..*, la classe Java `Studente` si differenzia nei seguenti aspetti:

1. Ha un'ulteriore funzione pubblica che realizza l'operazione `VerificaMoltMax()`, consentendo al cliente di verificare se sia possibile inserire un nuovo esame oppure no.

2. La funzione `int getLinkEsami()` lancia una opportuna eccezione (di tipo `RuntimeException`) quando l'oggetto di invocazione non rispetta il vincolo di molteplicità massima sull'associazione `Esame`.

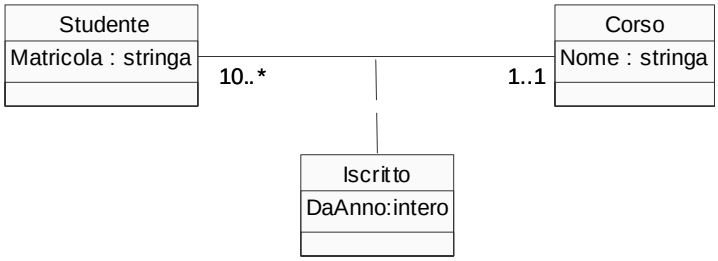
La classe Java Studente

```
// File ParteQuarta/MoltMaxConEccezioni/Studente.java

public class Studente implements SupportoAssocazioneEsame {
  private final String matricola;
  private Set insieme_link;
  static public final int MAX_LINK_ESAME = 28; // def. di costante
  public Studente(String n) {
    matricola = n;
    insieme_link = new InsiemeArrayOmogeneo(TipoLinkEsame.class);
  }

  public String getMatricola() { return matricola; }

  public boolean verificaMoltMax() {
    return insieme_link.size()<MAX_LINK_ESAME;
  }
```



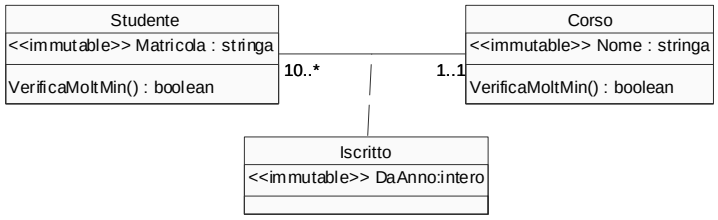
```
  public Iterator getLinkEsame() throws RuntimeException {
    if (verificaMoltMax() == false)
       throw new RunTimeException("Molteplicita' massima  violata");
    else return insieme_link.iterator();
  }

  public void inserisciLinkEsame(AssociazioneEsame a)  {
    if (a != null  &&
        a.getLink().getStudente() == this)
      insieme_link.add(a.getLink());
    else throw new RuntimeException("Inserimento link impossibile");
  }
  public void eliminaLinkEsame(AssociazioneEsame a) {
    if (a != null)
      insieme_link.remove(a.getLink());
    else throw new RuntimeException("Eliminazione link impossibile");
  }
}
```

Decimo caso: molteplicità minima > 0



A causa delle molteplicità, sia *Studente* sia *CorsoDiLaurea* hanno responsabilità sull'associazione.

Diagramma Realizzativo Decimo Caso



Per il controllo delle molteplicità minime, sia *Studente* sia *Corso* hanno due operazioni aggiuntive.

Molteplicità minima diversa da zero (cont.)

Questo caso è quello che meglio dimostra il fatto che imporre che tutti i vincoli di molteplicità di un diagramma delle classi siano rispettati *in ogni momento* è, in generale, molto complicato.

Nel nostro esempio, uno studente potrebbe nascere solamente nel momento in cui esiste già un corso di laurea, ma un corso di laurea deve avere almeno dieci studenti, e questo indice una intrinseca complessità nel creare oggetti, e al tempo stesso fare in modo che essi non violino vincoli di molteplicità minima. Problemi simili si hanno nel momento in cui i link vengono eliminati.

Come già detto, la strategia che seguiremo semplifica il problema, **ammettendo che gli oggetti possano essere in uno stato che non rispetta il vincolo di molteplicità minima, ma lanciando una eccezione** nel momento in cui un cliente chieda di utilizzare un link (relativo ad una associazione `A`) di un oggetto che non rispetta tale vincolo sull'associazione `A`.

Molteplicità minima diversa da zero (cont.)

- Rispetto al caso in cui il vincolo di molteplicità sia 0..*, la classe Java `CorsoDiLaurea` ha un'ulteriore funzione pubblica che realizza l'operazione `VerificaMoltMin`, verificando che il numero di studenti iscritti per il corso di laurea oggetto di invocazione sia almeno 10.

  In questa maniera, il cliente si può rendere conto se sia rispettato il vincolo oppure no. La funzione `int getLinkIscritto()` lancia una eccezione quando l'oggetto di invocazione non rispetta il vincolo di molteplicità minima sull'associazione `Iscritto`.

- Analoghe considerazioni valgono per la classe `Studente`. La classe `Studente` ha la funzione pubblica che realizza `VerificaMoltMin()`, restituendo `true` se lo studente partecipa alla associazione *Iscritto*, `false` altrimenti. Come nel caso precedente la funzione `getLinkIscritto()`: la funzione

lancia una eccezione se il vincolo di partecipazione obbligatoria non è rispettato.

Riportiamo il codice delle classi `CorsoDiLaurea`, e `Studente`.

La classe Java CorsoDiLaurea

```
// File ParteQuarta/MoltMinConEccezioni/CorsoDiLaurea.java

public class CorsoDiLaurea implements SupportoAssociazioneIscritto {
  private final String nome;
  private Set insieme_link;
  static public final int MIN_LINK_ISCRITTO = 10;
  public CorsoDiLaurea(String n) {
    nome = n;
    insieme_link = new InsiemeArrayOmmogeneo(TipoLinkIscritto.class);
  }
  public String getNome() { return nome; }

  public boolean verificaMoltMin () {
    return insieme_link.size()>=MIN_LINK_ISCRITTO;
  }
```

```
  public Iterator getLinkIscritto() {
    if (verificaMoltMin() == false)
      throw new RuntimeException("Cardinalita minima violata");
    else return insieme_link.iterator();
  }
  void inserisciLinkIscritto(AssociazioneIscritto a) {
    if (a != null  &&
        a.getLink().getCorsoDiLaurea() == this)
      insieme_link.add(a.getLink());
    else throw new RuntimeException("Inserimento impossibile");

  }
  void eliminaLinkIscritto(AssociazioneIscritto a) {
    if (a != null)
      insieme_link.remove(a.getLink());
     else throw new RuntimeException("Eliminazione impossibile");
 }
}
```

La classe Java Studente

```
// File ParteQuarta/MoltMinConEccezioni/Studente.java

public class Studente implements SupportoAssociazioneIscritto {
  private final String matricola;
  private TipoLinkIscritto link;
  public Studente(String n) { matricola = n; }
  public String getMatricola() { return matricola; }

  public boolean verificaMoltMin() { return link != null; }
  public TipoLinkIscritto getLinkIscritto() {
    if (verificaMolMin() == false)
      throw new RuntimeException("Cardinalita minima violata");
    else
      return link;
  }
  public void inserisciLinkIscritto(AssociazioneIscritto a) {
```

```
    if (a != null  && a.getLink().getStudente() == this &&
        link != null)
      link = a;
    else throw new RuntimeException("Inserimento link impossibile");

  }
  public void eliminaLinkIscritto(AssociazioneIscritto a) {
    if (a != null &&
        a.getLink().equals(link))
      link = null;
    else throw new RuntimeException("Eliminazione link impossibile");
  }
}
```

Associazioni n-arie

Si trattano generalizzando quanto visto per le associazioni binarie.

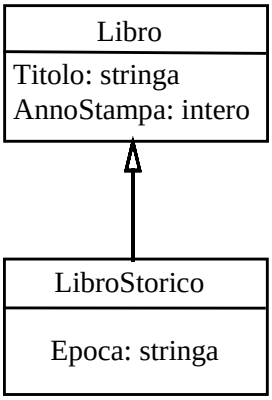
Ricordiamo che noi assumiamo che le molteplicità delle associazioni n-arie siano sempre 0..*.

In ogni caso, per un'associazione n-aria `A`, anche se non ha attributi, si definisce la corrispondente classe `TipoLinkA`.

Si assume (almeno in questo corso) che la responsabilità della associazione n-aria sia sempre di tutte le classi che partecipano alla stessa. Come per le associazioni binarie si definisce la classe `AssociazioneA`.

Generalizzazione

Affrontiamo ora il caso in cui abbiamo una generalizzazione nel diagramma delle classi. Ci referiamo al seguente esempio.



Generalizzazione (cont.)

1. La superclasse UML (*Libro*) diventa una classe base Java (`Libro`), e la sottoclasse UML (*LibroStorico*) diventa una classe derivata Java (`LibroStorico`).

   Infatti, poiché ogni istanza di *LibroStorico* è anche istanza di *Libro*, vogliamo:

   - poter usare un oggetto della classe `LibroStorico` ogni volta che è lecito usare un oggetto della classe `Libro`, e
   - dare la possibilità agli utenti della classe `LibroStorico` di usare le funzioni pubbliche di `Libro`.

Generalizzazione (cont.)

2. Poiché ogni proprietà della classe *Libro* è anche una proprietà del tipo *LibroStorico*, in `Libro` tutto ciò che si vuole ereditare è **protetto**.

   Si noti che la possibilità di utilizzare la parte protetta di `Libro` implica che il progettista della classe `LibroStorico` (e delle classi eventualmente derivate da `LibroStorico`) deve avere una buona conoscenza dei metodi di rappresentazione e delle funzioni della classe `Libro`.

3. Nella classe `LibroStorico`:

   - ci si affida alla definizione di `Libro` per quelle proprietà (ad es., *AnnoStampa*, *Titolo*) che sono identiche per gli oggetti della classe `LibroStorico`;
   - si definiscono tutte le proprietà (dati e funzioni) che gli oggetti di *LibroStorico* hanno in più rispetto a quelle ereditate da *Libro* (ad es., *Epoca*).

Information hiding: riassunto

Fino ad ora abbiamo seguito il seguente approccio per garantire un alto livello di information hiding nella realizzazione di una classe UML *C* mediante una classe Java `C`:

- gli attributi di *C* corrispondono a campi **privati** della classe Java `C`;
- le operazioni di *C* corrispondono a campi **pubblici** di `C`;
- sono **pubblici** anche i costruttori di `C` e le funzioni `get` e `set`;
- sono invece **private** eventuali funzioni che dovessero servire per la realizzazione dei metodi della classe *C* (ma che non vogliamo rendere disponibili ai clienti), e i campi dati per la realizzazione di associazioni;
- tutte le classi Java sono **nello stesso package** (senza nome).

Information hiding e generalizzazione

Nell'ambito della realizzazione di generalizzazioni, è più ragionevole che i campi di `C` che non vogliamo che i clienti possano vedere siano **protetti**, e non privati.

Infatti, in questa maniera raggiungiamo un duplice scopo:

1. continuiamo ad impedire ai clienti generici di accedere direttamente ai metodi di rappresentazione e alle strutture di dati, mantenendo così alto il livello di information hiding;

2. diamo tale possibilità ai progettisti delle classi derivate da `C` (che non devono essere considerati clienti qualsiasi) garantendo in tal modo maggiore efficienza.

Information hiding e generalizzazione (cont.)

Occorre tuttavia tenere opportunamente conto delle regole di visibilità di Java, che garantiscono **maggiori diritti** ad una classe di uno stesso package, rispetto ad una classe derivata, ma di package diverso.

Non possiamo più, quindi, prevedere un solo package per tutte le classi Java, in quanto sarebbe vanificata la strutturazione in parte pubblica e parte protetta, poiché tutte le classi (anche quelle non derivate) avrebbero accesso ai campi protetti.

Da ciò emerge la necessità di prevedere **un package diverso** per ogni classe che ha campi protetti (tipicamente, ciò avviene quando fa parte di una gerarchia).

Generalizzazione e strutturazione in package

In particolare, seguiremo le seguenti regole:

- per ogni classe UML *C* che ha campi protetti prevediamo un package dal nome `C`, realizzato nel direttorio `C`, che contiene solamente il file dal nome `C.java`;

- ogni classe Java `D` che deve accedere ai campi di `C` conterrà la dichiarazione

  ```
  import C.*;
  ```


La classe Java Libro

```
// File ParteQuarta/Generalizzazione/Libro/Libro.java

package Libro;

public class Libro {
  protected final String titolo;
  protected final int annoStampa;
  public Libro(String t, int a) { titolo = t; annoStampa = a;}
  public String getTitolo() { return titolo; }
  public int getAnnoStampa() { return annoStampa; }
  public String toString() {
    return titolo + ", dato alle stampe nel " + annoStampa;
  }
}
```

La classe Java LibroStorico

```
// File ParteQuarta/Generalizzazione/LibroStorico/LibroStorico.java

package LibroStorico;
import Libro.*;

public class LibroStorico extends Libro {
  protected final String epoca;
  public LibroStorico(String t, int a, String e) {
    super(t,a);
    epoca = e;
  }
  public String getEpoca() { return epoca; }
  public String toString() {
    return super.toString() + ", ambientato nell'epoca: " + epoca;
  }
}
```


Esempio di cliente

InizioSpecificaUseCase Valutazione Biblioteca

QuantiAntichi (*i: Insieme(Libri), a: intero): intero*
pre: nessuna
post: *result* è il numero di libri dati alle stampe prima dell'anno *a* nell'insieme di libri *i*

QuantiStorici (*i: Insieme(Libri)): intero*
pre: nessuna
post: *result* è il numero di libri storici nell'insieme di libri *i*

FineSpecifica

Realizzazione del cliente

```
// File ParteQuarta/Generalizzazione/ValutazioneBiblioteca.java
import Libro.*;
import LibroStorico.*;

public class ValutazioneBiblioteca {
  public static int quantiAntichi(Set ins, int anno) {
    int quanti = 0;
    Iterator it = ins.iterator();
    while (ins.hasNext()) {
      Libro elem = (Libro)it.next();
      if (elem.getAnnoStampa() < anno)
        quanti++;
    }
    return quanti;
  }
  public static int quantiStorici(Set ins) {
```

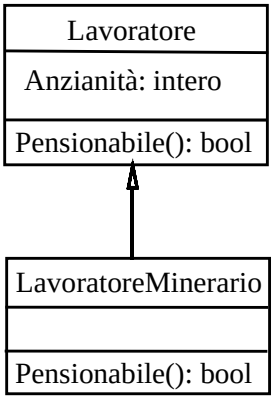


```
    int quanti = 0;
    Iterator it = ins.iterator();
    while (it.hasNext()) {
      Libro elem = (Libro)it.next();
      if (elem.getClass().equals(LibroStorico.class))
        quanti++;
    }
    return quanti;
  }
} // ValutazioneBiblioteca
```

Ridefinizione

Nella classe Java derivata si **ridefinisce** una funzione `F()` già definita nella classe base ogni volta che `F()`, quando viene eseguita su un oggetto della classe derivata, deve compiere operazioni diverse rispetto a quelle della classe base, ad esempio operazioni che riguardano le proprietà specifiche che la classe derivata possiede rispetto a quelle definite per quella base.

Ridefinizione: esempio



I lavoratori sono pensionabili con un'anzianità di 30 anni.
I lavoratori minerari sono pensionabili con un'anzianità di 25 anni.

Ridefinizione: esempio (cont.)

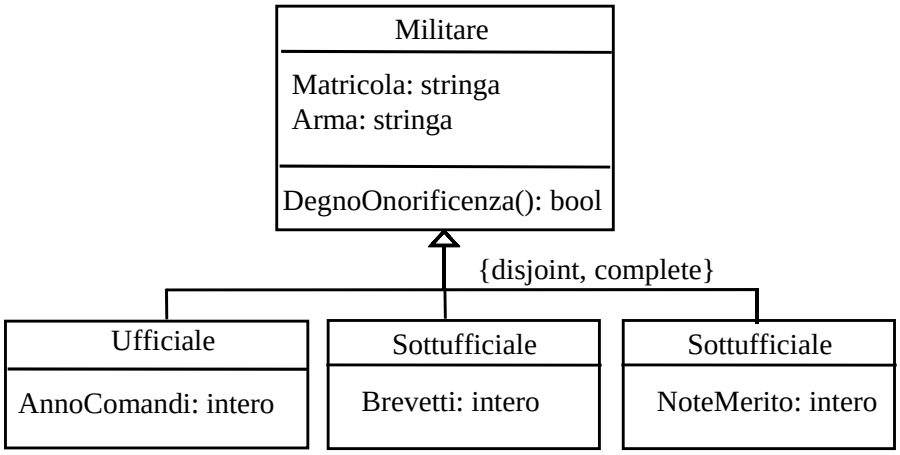
```
// File ParteQuarta/Generalizzazione/Lavoratore/Lavoratore.java

package Lavoratore;

public class Lavoratore {
  protected int anzianita;
  public int getAnzianita() { return anzianita; }
  public void setAnzianita(int a) { anzianita = a; }
  public boolean pensionabile() { return anzianita > 30; }
}
```


Ridefinizione: esempio (cont.)

```
// File ParteQuarta/Generalizzazione/LavoratoreMinerario/...
// ...LavoratoreMinerario.java

package LavoratoreMinerario;
import Lavoratore.*;

public class LavoratoreMinerario extends Lavoratore {
  public boolean pensionabile() { return anzianita > 25; }
    // OVERRIDING
}
```

Generalizzazioni disgiunte e complete

Siccome Java non supporta l'ereditarietà multipla, **assumiamo che ogni generalizzazione sia disgiunta** (ciò può essere ottenuto mediante opportune trasformazioni, come descritto nella parte del corso dedicata all'analisi). Quando la generalizzazione è anche completa, occorre fare delle considerazioni ulteriori, come mostrato da questo esempio.



Generalizzazioni disgiunte e complete (cont.)

Il diagramma delle classi ci dice che non esistono istanze di *Militare* che non siano istanze di almeno una delle classi *Ufficiale*, *Sottufficiale* o *MilitareDiTruppa*.

Per questo motivo la classe Java `Militare` deve essere una `abstract class`. La definizione di `Militare` come classe base astratta consente di progettare clienti che astraggono rispetto alle peculiarità delle sue sottoclassi.

In questo modo, infatti, **non si potranno definire oggetti che sono istanze dirette della classe** `Militare`.

Viceversa, le classi Java `Ufficiale`, `Sottufficiale` e `MilitareDiTruppa` saranno classi non `abstract` (a meno che siano anch'esse superclassi per generalizzazioni disgiunte e complete).

Funzioni Java non astratte

Alcune proprietà della classe UML *Militare*, come ad esempio l'attributo "arma di appartenenza", sono dettagliabili completamente al livello della classe stessa.

La gestione di queste proprietà verrà realizzata tramite funzioni non `abstract` della classe Java `Militare`.

Funzioni Java astratte

Tra le operazioni che associamo a *Militare* ve ne possono essere invece alcune che sono dettagliabili **solo quando vengono associate** ad una delle sottoclassi.

Ad esempio, l'operazione che determina se un militare è degno di onoreficenza potrebbe dipendere da parametri relativi al fatto se esso è ufficiale, sottufficiale oppure di truppa. L'operazione *DegnoDiOnoreficenza* si può associare alla classe *Militare* solo concettualmente, mentre il calcolo che essa effettua si può rappresentare in modo preciso solo al livello della sottoclasse.

La corrispondente funzione Java verrà **dichiarata** come `abstract` nella classe `Militare`. La sua **definizione** viene demandata alle classi java `Ufficiale`, `Sottufficiale` o `MilitareDiTruppa`.

Esempio: Militare e sottoclassi

Assumiamo che, per le sottoclassi di *Militare*, i criteri per essere degni di onoreficenza siano i seguenti:

Ufficiale: avere effettuato più di dieci anni di comando.

Sottufficiale: avere conseguito più di quattro brevetti di specializzazione.

MilitareDiTruppa: avere ricevuto più di due note di merito.

La classe astratta Java Militare

```
// File ParteQuarta/Generalizzazione/Militare/Militare.java

package Militare;

public abstract class Militare {
  protected String arma;
  protected String matricola;
  public Militare(String a, String m) { arma = a; matricola = m; }
  public String getArma() { return arma; }
  public String getMatricola() { return matricola; }
  abstract public boolean degnoDiOnoreficenza();
  public String toString() {
    return "Matricola: " + matricola + ". Arma di appartenenza: " + arma;
  }
}
```

Un cliente della classe astratta

```
  public static void stampaStatoDiServizio(Militare mil) {
    System.out.println("========== FORZE ARMATE ========== ");
    System.out.println("STATO DI SERVIZIO DEL MILITARE");
    System.out.println(mil);
    if (mil.degnoDiOnoreficenza())
      System.out.println("SI E' PARTICOLARMENTE DISTINTO IN SERVIZIO");
  }
```


La classe Java Ufficiale

```
// File ParteQuarta/Generalizzazione/Ufficiale/Ufficiale.java

package Ufficiale;
import Militare.*;

public class Ufficiale extends Militare {
  protected int anni_comando;
  public Ufficiale(String a, String m) { super(a,m); }
  public int getAnniComando() { return anni_comando; }
  public void incrementaAnniComando() { anni_comando++; }
  public boolean degnoDiOnoreficenza() {
    return anni_comando > 10;
  }
}
```

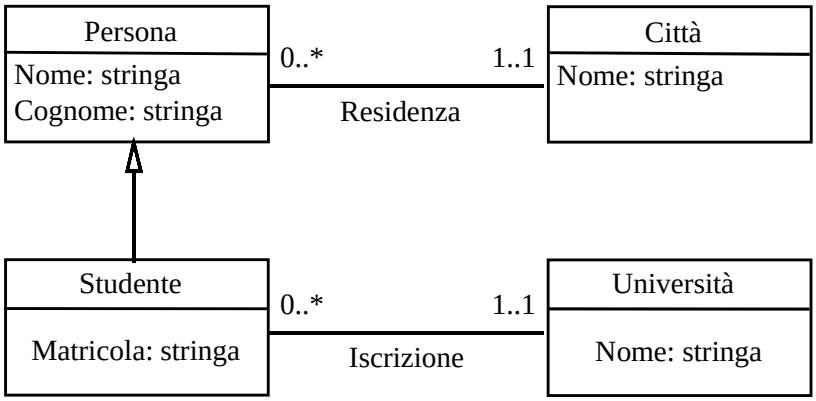
La classe Java Sottufficiale

```
// File ParteQuarta/Generalizzazione/Sottufficiale/Sottufficiale.java

package Sottufficiale;
import Militare.*;

public class Sottufficiale extends Militare {
  protected int brevetti_specializzazione;
  public Sottufficiale(String a, String m) { super(a,m); }
  public int getBrevettiSpecializzazione() {
    return brevetti_specializzazione; }
  public void incrementaBrevettiSpecializzazione() {
    brevetti_specializzazione++; }
  public boolean degnoDiOnoreficenza() {
    return brevetti_specializzazione > 4;
  }
}
```


La classe Java MilitareDiTruppa

```
// File ParteQuarta/Generalizzazione/MilitareDiTruppa/MilitareDiTruppa.java

package MilitareDiTruppa;
import Militare.*;

public class MilitareDiTruppa extends Militare {
  protected int note_di_merito;
  public MilitareDiTruppa(String a, String m) { super(a,m); }
  public int getNoteDiMerito() { return note_di_merito; }
  public void incrementaNoteDiMerito() { note_di_merito++; }
  public boolean degnoDiOnoreficenza() {
    return note_di_merito > 2;
  }
}
```

Esercizio



Realizzare in Java questo diagramma delle classi. Scrivere una funzione cliente che, data un'università, restituisca la città da cui proviene la maggior parte dei suoi studenti.

Esercizio

Arricchire il diagramma delle classi precedente, in maniera che tenga conto del fatto che ogni università ha sede in una città, e che una città può avere un numero qualsiasi di università.

Uno studente si definisce *locale* se è iscritto ad un'università della città in cui risiede. Scrivere una funzione cliente che stabilisca se uno studente è locale oppure no.

Molteplicità 0..1 di attributi

Quando un attributo (di tipo T) di una classe o di una associazione UML ha molteplicità 0..1, dobbiamo prevedere che, per ogni istanza (della classe o della associazione), il valore ad essa associata dall'attributo sia rappresentato in realtà mediante due elementi:

- un valore booleano, che vale true quando esista effettivamente il valore associato dall'attributo, e false quando invece tale valore non esista;
- un valore di tipo T, che è significativo, e rappresenta proprio il valore associato all'istanza dall'attributo, solo quando il valore booleano ad esso associato è true, e non è invece significativo nel caso contrario.

Esercizio: Realizzare la classe *Persona*, in maniera che ogni persona possa non avere un indirizzo noto.

Molteplicità massima > 1 di attributi

Quando un attributo di una classe o di una associazione UML ha molteplicità massima > 1, in particolare *, (ad es., *NumTel: int* $\{0..*\}$), possiamo semplicemente usare una classe contenitore apposita per la rappresentazione dei valori che l'attributo associa ad un oggetto.

Attenzione: se si usano classi contenitore (come `Set` o `InsiemeSS`) che vogliono oggetti di una classe (e non valori di un tipo base) come elementi, occorre usare opportune classi (ad esempio `Integer` al posto di `int`).

Esercizio: Realizzare la classe *Persona*, in maniera che ogni persona possa avere un numero qualsiasi di numeri di telefono.

Realizzazione di tipi UML

Abbiamo già incontrato alcuni *tipi astratti* (ad es., *Insieme*, *Vettore*).

Daremo alcune regole per realizzare tipi UML tramite classi Java.

`toString()`: si può prevedere di farne overriding, per avere una rappresentazione testuale dell'oggetto.

`equals()`: **è necessario** fare overriding della funzione `equals()` ereditata dalla classe `Object`.

Infatti due valori sono uguali solo se sono lo stesso valore, e quindi il comportamento di default della funzione `equals()` non è corretto.

Realizzazione di tipi UML (cont.)

`clone()`: ci sono due possibilità:

1. Nessuna funzione della classe Java effettua side-effect. In questo caso, `clone()` non si ridefinisce (gli oggetti sono *immutabili*.)

2. Qualche funzione della classe Java effettua side-effect. In questo caso, poiché i moduli clienti hanno tipicamente la necessità di copiare valori (ad esempio, se sono argomenti di funzioni) **si mette a disposizione la possibilità di copiare un oggetto**, rendendo disponibile la funzione `clone()` (facendo overriding della funzione `protected` ereditata da `Object`).

Esempio: la classe Java Data

```
// File ParteQuarta/Data.java
class Data implements Cloneable {
  public Data() {
    giorno = 1;
    mese = 1;
    anno = 2000;
  }
  public Data(int a, int me, int g) {
    giorno = g;
    mese = me;
    anno = a;
    if (!valida()) {
      giorno = 1;
      mese = 1;
      anno = 2000;
    }
```



```
  }
  public int giorno() {
    return giorno;
  }
  public int mese() {
    return mese;
  }
  public int anno() {
    return anno;
  }
  public boolean prima(Data d) {
    return ((anno < d.anno)
            ||  (anno == d.anno && mese < d.mese)
            ||  (anno == d.anno && mese == d.mese && giorno < d.giorno));
  }
  public void avanzaUnGiorno() {
    // FA SIDE-EFFECT
    if (giorno == giorniDelMese())
```

```
    if (mese == 12) {
      giorno = 1;
      mese = 1;
      anno++;
    }
    else {
      giorno = 1;
      mese++;
    }
  else
    giorno++;
}
public String toString() {
  return giorno + "/" + mese + "/" + anno;
}
public Object clone() {
  try {
    Data d = (Data)super.clone();
```

```
    return d;
  } catch (CloneNotSupportedException e) {
  // non puo' accadere, ma va comunque gestito
    throw new InternalError(e.toString());
  }
}
public boolean equals(Object o) {
  if (o != null && getClass().equals(o.getClass())) {
    Data d = (Data)o;
    return d.giorno == giorno && d.mese == mese && d.anno == anno;
  }
  else return false;
}

private int giorno, mese, anno;

private int giorniDelMese() {
  switch (mese) {
```

```
    case 2:
      if (bisestile()) return 29;
      else return 28;
    case 4: case 6: case 9: case 11: return 30;
    default: return 31;
    }
  }
  private boolean bisestile() {
    return ((anno % 4 == 0) && (anno % 100 != 0))
      || (anno % 400 == 0);
  }
  private boolean valida() {
    return anno > 0 && anno < 3000
      && mese > 0 && mese < 13
      && giorno > 0 && giorno <= giorniDelMese();
  }
}
```

Use cases

Come abbiamo avuto più volte maniera di vedere, uno use case *U* si realizza in Java nel seguente modo:

- una classe Java `U`, tipicamente da sola in un file `U.java`;
- una funzione `static` di `U` per ogni operazione di *U*.

Non siamo interessati infatti ad avere oggetti di `U`. Questa classe è un mero **contenitore di funzioni**.

Organizzazione in packages

- Si definisce un package Java `P` (ed un corrispondente direttorio) per tutta l'applicazione (al limite `P` può essere il package senza nome)

- Si inserisce nel direttorio del package `P` un file per ogni classe (sia proveniente dal diagramma delle classi, sia proveniente dal diagramma degli use case) senza campi `protected`, con la definizione di tale classe, dichiarando che essa appartiene al package `P`

- Si definisce un sottopackage di `P` (e quindi un corrispondente sottodirettorio nel direttorio del package `P`) per ogni classe con almeno un campo `protected`, e si inserisce in esso il file con la definizione di tale classe, dichiarando, dichiarando che essa appartiene al sottopackage