

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

1

## 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 **specific**a
  - una specifica per ogni classe
  - una specifica per ogni use case

2

## 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

3

- 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 programmatiche.

Nota: il diagramma delle classi realizzativo è **peggiore** del diagramma delle classi concettuale, in quanto si sono effettuate delle scelte influenzate da tecnicità 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**.

4

## Diagramma delle classi realizzativo

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

Persona
<<immutable>> Nome : stringa
<<immutable>> Cognome : stringa
<<immutable>> Nascita : data
<<mutable>> Coniugato : boolean

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

Una volta stabilito questo possiamo passare alla realizzazione del codice.

6

## Singola classe UML con soli attributi

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

Persona
Nome : stringa
Cognome : stringa
Nascita : data
Coniugato : boolean

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

5

## 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.

7

## 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 creazione 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).

8

## 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., **Num-Tel**: **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**).

9

## 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**.

10

## 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.

11

## 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;
    }
}
```

12

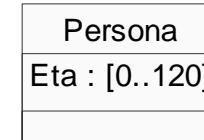
```
public String getCognome() {
    return cognome;
}
public int getGiornoNascita() {
    return giorno_nascita;
}
public int getMeseNascita() {
    return mese_nascita;
}
public int getAnnoNascita() {
    return anno_nascita;
}
public void setConiugato(boolean c) {
    coniugato = c;
}
public boolean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

13

## 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
```

14

```
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.

15

16

## 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 {
```

17

```
        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);
    }
}
```

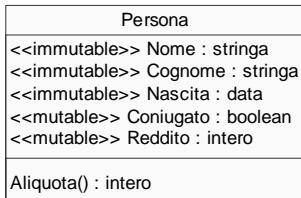
## Classe UML con attributi e operazioni

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.

18

## Classe UML con attributi e operazioni: esempio



### InizioSpecificazioneClasse 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.

### FineSpecificazioneClasse Persona

19

```
}

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;
}
```

## 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 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 + "/" +
    }
}
```

20

```

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

```

## Esempio di cliente

### InizioSpecificUseCase Analisi Statistica

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

pre: nessuna

post: *result* è il numero di coniugati nell'insieme di persone *i*

### FineSpecific

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;
    }
}

```

21

## Esercizio 1: classi UML con operazioni

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

### InizioSpecificClasse *Persona*

**Aliquota** (): intero ...

**Età** (*d*: data): intero

pre: nessuna

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

### FineSpecific

## 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() {

```

22

23

```

        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.

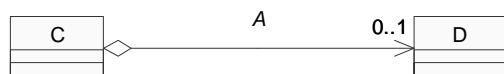
Diciamo che **una classe C ha responsabilità sull'associazione A**, quando, per ogni oggetto  $x$  che è istanza di  $C$  vogliamo poter eseguire opportune operazioni sulle istanze di  $A$  a cui  $x$  partecipa, che hanno lo scopo di:

- 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  $A$ .

26

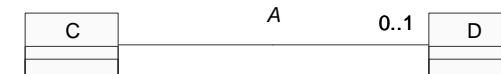
## Diagramma delle classi realizzativo



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  $D$ .

28

## 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 rappresentazione dei link nella associazione  $A$ .

27

## Realizzazione delle associazioni: primo caso

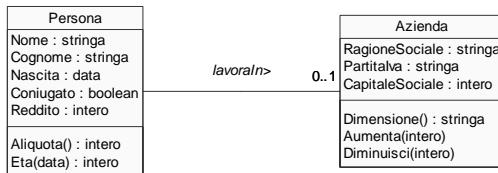
In questo caso, la realizzazione del codice è simile a quella per un attributo. Infatti, oltre a quanto stabilito per gli attributi e le operazioni, per ogni associazione  $A$  del tipo mostrato in figura, aggiungiamo alla classe Java  $C$

- 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  $A$ .

29

## Due classi legate da associazione: esempio

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

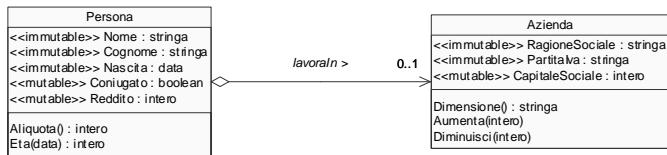


Supponiamo che solo *Persona* abbia responsabilità sull'associazione (non ci interessa conoscere i dipendenti di un'azienda, ma solo in quale azienda lavora una persona che lavora), assumiamo inoltre di non volere mantenere la rappresentazione esplicita dei link della associazione.

30

## 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.

32

## Specificazione della classe UML Azienda

### InizioSpecificazioneClasse 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*

**Diminisci (i: intero)**

pre:  $1 \leq i \leq this.CapitaleSociale$

post: *this.CapitaleSociale* vale *pre(this.CapitaleSociale) - i*

### FineSpecificazione

31

## 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() {
```

33

```

        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*:

### InizioSpecificheUseCase 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*

### FineSpecifiche

---

Rappresentare l'insieme in input semplicemente come un vettore.

## Soluzione esercizio 3

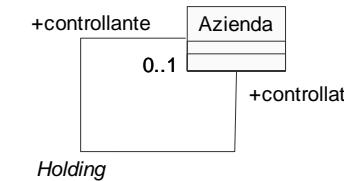
```
// File ParteQuarta/Associazioni01/Esercizio3.java

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;
    }
}
```

36

## Realizzazione delle associazioni: secondo caso

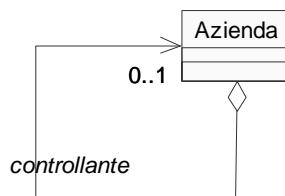
Quanto detto vale anche per il caso in cui l'associazione coinvolga più volte la stessa classe. In questo caso il concetto di responsabilità si attribuisce ai ruoli, piuttosto che alla classe.



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.

37

## 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*).

38

## 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;
    }
}
```

39

## 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 **representazione 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 attributi di molteplicità 1..1.

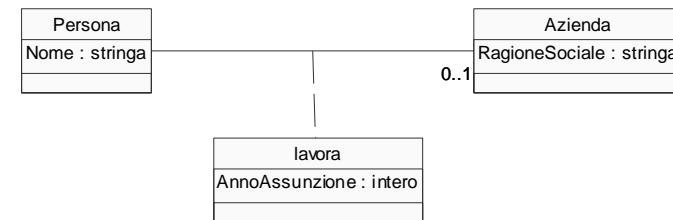
Rimangono le seguenti assunzioni:

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

40

## Realizzazione delle associazioni: terzo caso

Esempio

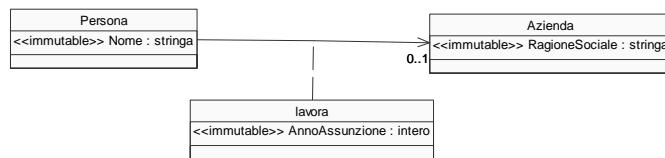


Solo *Persona* ha responsabilità sull'associazione, vogliamo mantenere una rappresentazione esplicita dei link per gestire opportunamente gli attributi dell'associazione.

41

## 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.

42

## 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.

43

## 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 **TypoLinkAs**, 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**).

44

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

## Realizzazione in Java della classe TypoLinkLavora

```
// File ParteQuarta/Ass01Attr/TypoLinkLavora.java  
  
public class TypoLinkLavora {  
    private final Persona laPersona;  
    private final Azienda laAzienda;  
    private final int annoAssunzione;  
  
    public TypoLinkLavora(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) {
```

45

## Realizzazione in Java della classe Persona

```
// File ParteQuarta/Ass01Attr/Persona.java  
  
public class Persona {  
    private final String nome;  
    private TypoLinkLavora link;  
    public Persona(String n) { nome = n; }  
    public String getNome() { return nome; }  
    public void inserisciLinkLavora(TypoLinkLavora t) {  
        if (link == null && t != null &&  
            t.getAzienda() != null && t.getPersona() == this)  
            link = t;  
    }  
    public void eliminaLinkLavora() {  
        link = null;  
    }  
    public TypoLinkLavora getLinkLavora() { return link; }  
}
```

46

## Considerazioni sulle classi Java

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

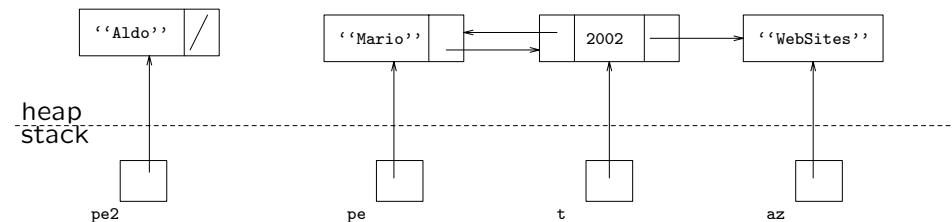
Di fatto un oggetto della classe è *immutable*, ovvero una volta creato non può più essere cambiato.

- 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.

47

## Possibile stato della memoria



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

48

## 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");  
  
    TypoLinkLavora t = new TypoLinkLavora(az,pe,2002);  
  
    pe.inserisciLinkLavora(t);  
}
```

49

## Esercizio 4: cliente

Realizzare in Java lo use case *Ristrutturazione Industriale*:  
InizioSpecificoUseCase **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*

FineSpecifico

---

Rappresentare l'insieme in input semplicemente come un vettore.

50

## 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);
            }
        }
    }
}
```

51

## Metodi naive di realizzazione

Se abbiamo una associazione con attributi ma non siamo interessati alla rappresentazione esplicita dei link possiamo 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 sì 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; }
}
```

52

```
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.

53

## 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.

54

## 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*).

55

## 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*.

56

## 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 Programme-r's Interface*), ovvero i servizi forniti.

57

## 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.

58

## 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.

59

## 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.

60

## 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()
```

61

```

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

```

```

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;
    }
}

```

## 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 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);
}

```

```

// 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;

```

```

        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;
}

```

```

    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**.

63

## 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.

64

## 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.

65

## 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.

66

## 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).

67

## Diagramma Realizzativo: Quarto Caso



68

## Realizzazione in Java della classe Persona

```
// File ParteQuarta/Ass0STAR/Persona.java
import java.util.*;
import insiemarray.*;

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);
    }
}
```

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

69

## Classe Java Persona: considerazioni

- La classe ha un campo dati di tipo Set.
- Il costruttore della classe Persona crea un oggetto della classe InsiemeArrayOmo 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.

70

## 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.

71

## Esempio di cliente

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

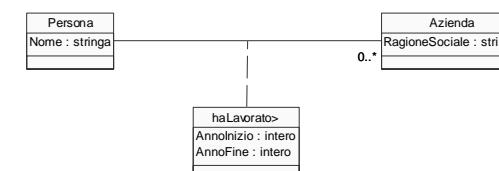
```
public static void stampaAziende(Persona p) {
    System.out.println("La persona " + p.getNome() +
        " ha lavorato nelle seguenti aziende:");
    Iterator it = p.getHaLavorato();
    while (it.hasNext()) {
        Azienda az = (Azienda)it.next();
        System.out.println(az.getRagioneSociale());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

72

## Quinto caso: associazioni 0..\* con attributi

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

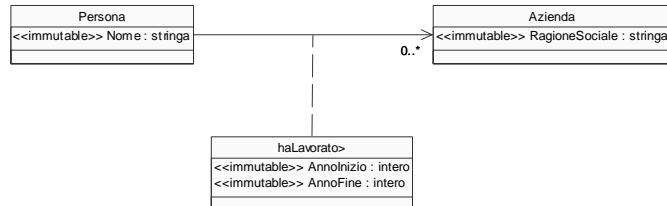
Schema concettuale da realizzare in Java (solo la classe Persona ha responsabilità sull'associazione):



Assumiamo per semplicità che si lavori sempre per anni interi.

73

## Diagramma Realizzativo: Quinto Caso



74

## 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.

75

## 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.

76

## 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())) {
```

77

```

        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.

Sceglieremo 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.

78

## La classe Java Persona

```

// File ParteQuarta/Ass0STARAttr/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();
}
}

```

79

## 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());  
    }  
}
```

80

## Esercizio 5: cliente

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

**FineSpecifico**

81

## Soluzione esercizio 5

```
// File ParteQuarta/AssOSTARAttr/AnalisiMercatoLavoro.java  
import java.util.*;  
  
public class AnalisiMercatoLavoro {  
    public static int periodoPiùLungo(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()) {
```

82

```
        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();
}

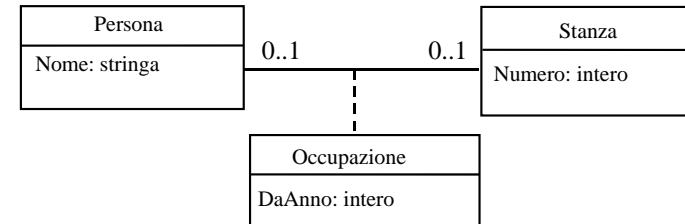
public static void analisi(Persona p) {
    System.out.println("Il periodo piu' lungo in cui " + p.getNome() +
                       " ha lavorato per la stessa azienda e' di anni: " +
                       periodoPiùLungo(p));
    System.out.println(p.getNome() + (riAssuntoSubito(p)?" ":" non ") +
                       "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.



83

## 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.

84

## 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*.

85

## 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.

86

## 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 &&
            a.getLink().getLinkOccupazione() == null)
```

87

```
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 &&
            a.getLink().getLinkOccupazione() == null)
            link = a.getLink();
    }
}
```

88

```

        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.

89

## 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;
    }
}

```

90

## 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**.

91

## 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().eliminaLinkOccupazione(k);
            y.getPersona().eliminaLinkOccupazione(k);
        }
    }

    public static void elimina(TipoLinkOccupazione y) {
```

92

```
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()`.

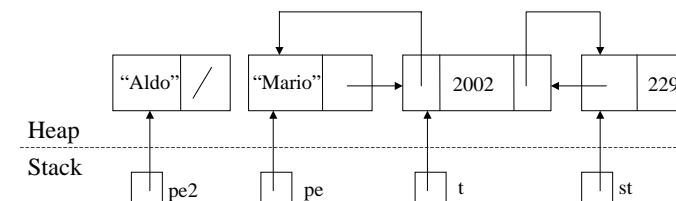
93

## 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`.

94

## 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.

95

## 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);  
}
```

96

## Esercizio 6: cliente

Realizzare in Java lo use case *Riallocazione Personale*:

### InizioSpecificoUseCase 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

...

97

## 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*

### FineSpecifiche

98

```
    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
```

## Soluzione esercizio 6

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

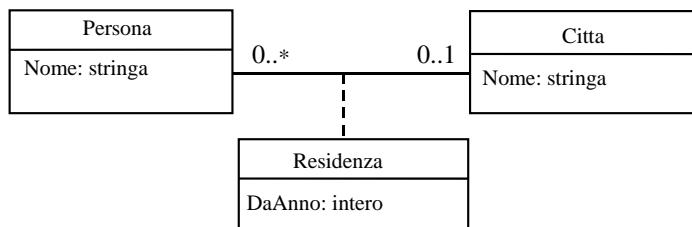
public class RiallocazionePersonale {
    public static void promuovi(Set ins, Stanza st, int anno) {
        // alla persona di ins che è da più tempo nella
        // stessa stanza viene assegnata la stanza st,
        // a partire dall'anno passato come argomento
        Iterator it = ins.iterator();
        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);
            }
        }
    }
}
```

99

```
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).

100

## 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.

101

- infine implementa l'interfaccia *SupportoAssociazioneResidenza* realizzando le funzioni (*inserisciLinkResidenza()*, *eliminaLinkResidenza()* per la gestione dell'insieme dei link.

### La classe Java Citta

```
// File ParteQuarta/RespEntrambi0STAR/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() {
```

102

```

        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("Elimina 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.

103

## 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!");
}

```

104

## La classe Java TipoLinkOccupazione

```
// File ParteQuarta/RespEntrambi0STAR/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 &&
```

105

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

106

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

### FineSpecificia

107

## Soluzione esercizio 7

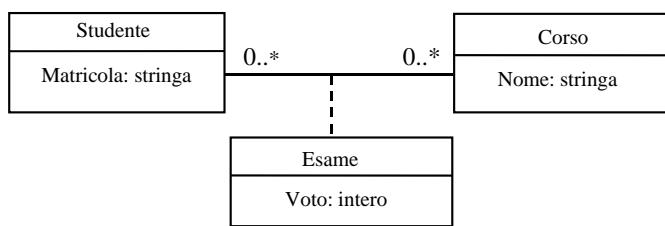
```
// File ParteQuarta/RespEntrambiOSTAR/GestioneAnagrafe.java
import java.util.*;
import insiemearray.*;

public class GestioneAnagrafe {
    public static Set trovaNuovi(Città 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;
    }
}
```

108

## 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.

109

## 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 *InsiemeSS*, per poter rappresentare tutti i link;

l'oggetto di classe *InsiemeSS* viene creato tramite il costruttore;

- hanno tre campi funzione (*inserisciLinkEsame()*, *eliminaLinkEsame()* e *getLinkEsame()*) per la gestione dell'insieme dei link;

quest'ultima restituisce **una copia** dell'insieme dei link.

Riportiamo il codice di tutte le classi.

110

## 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();
    }
}
```

111

```
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/RespEntrambiOSTAR2/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();
    }
}
```

112

```
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 &&
```

113

```
        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);
```

114

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

## Esercizio 8: cliente

Realizzare in Java lo use case *Valutazione Didattica*:

### InizioSpecificUseCase 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

### FineSpecific

115

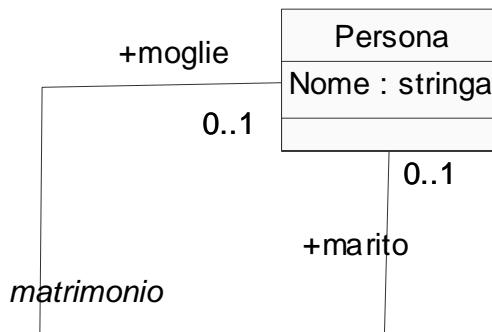
## Soluzione esercizio 8

```
// File ParteQuarta/RespEntrambiOSTAR2/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;
    }
}
```

116

## Caso Particolare



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

117

## 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 **AssociazioneMatrimonio** faranno uso di entrambe le interfacce.

118