SAPIENZA Università di Roma, Facoltà di Ingegneria

Corso di
PROGETTAZIONE DEL SOFTWARE
Laurea in Ingegneria Informatica
Prof. Giuseppe De Giacomo
A.A. 2010/11

LA FASE DI REALIZZAZIONE

(Diagrammi delle classi)

La fase di realizzazione

La fase di realizzazione si occupa di:

- scrivere il **codice** del programma, e
- produrre parte della **documentazione**

Il suo input è costituito da:

- l'output della fase di **analisi**, e
- l'output della fase di **progetto**.

Traduzione in Java del diagramma delle classi

Nell'esposizione di questo argomento, seguiremo quest'ordine:

1. **realizzazione di singole classi**,
2. **realizzazione delle associazioni**,
3. **realizzazione delle generalizzazioni**.

Realizzazione di una classe UML con soli attributi

Assumiamo, per il momento, che la molteplicità di tutti gli attributi sia 1..1.

- Gli attributi della classe UML diventano campi privati (o protetti) della classe Java, gestiti da opportune funzioni pubbliche:
  - la funzione **get** serve a restituire al cliente il valore dell'attributo;
  - la funzione **set** consente al cliente di cambiare il valore dell'attributo.
- I tipi Java per gli attributi vanno scelti secondo la *tabella di corrispondenza dei tipi* UML prodotta durante la fase di progetto.

Realizzazione di una classe UML con soli attributi (cont.)

- Si sceglie un opportuno valore iniziale per ogni attributo:
  - affidandosi al valore di default di Java, oppure
  - fissandone il valore nella dichiarazione (se tale valore iniziale va bene per tutti gli oggetti), oppure
  - facendo in modo che il valore iniziale sia fissato, oggetto per oggetto, mediante un costruttore.
- Per quegli attributi per i quali non ha senso prevedere di cambiare il valore (secondo la *tabella delle proprietà immutabili* prodotta durante la fase di progetto), non si definisce la corrispondente funzione **set**.

Metodologia per la realizzazione

Da classe UML *C* a classe Java `C`.

- La classe Java `C` è `public` e si trova in un file dal nome `C.java`.
- `C` è derivata da `Object` (no `extends`).

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.
- Tali campi possono essere dichiarati `final`, se non vengono più cambiati dopo la creazione dell'oggetto (secondo la *tabella delle proprietà immutabili* prodotta nella fase di progetto).

  Nota: dichiarando `final` *un campo dati si impone che esso non possa essere modificato dopo l'inizializzazione. Ma se il campo dati contiene è un riferimento ad un oggetto nulla impedisce di modificare l'oggetto stesso. Quindi l'efficacia di* `final` *è limitata.*

Metodologia per la realizzazione: campi funzio

I campi funzione della classe `C` sono tutti `public`.

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`: in generale, vanno previste per tutti i campi dati.

Funzioni `set`: vanno previste solo per quei campi dati che possono mutare (tipicamente, non dichiarati `final`).

Metodologia: funzioni speciali

`equals()`: tipicamente, **non è necessario** 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.

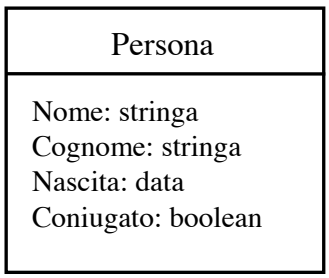
`clone()`: in molti casi, è ragionevole decidere di **non mettere a disposizione la possibilità di copiare un oggetto**, e non rendere disponibile la funzione `clone()` (non facendo overriding della funzione `protected` ereditata da `Object`).

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.

Singola classe UML con soli attributi: esempio

Risultato fase di analisi:



Risultato fase di progetto:

Tipo UML	**Rappresentazione in** Java
stringa	`String`
data	`int,int,int`
booleano	`boolean`

Classe UML	**Proprietà immutabile**
Persona	*nome*
	cognome
	nascita

	Proprietà	
Classe UML	**nota alla nascita**	**non nota alla nascita**

Per default, una persona non è coniugata.

Realizzazione in Java

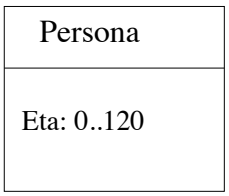
```
// File SoloAttributi/Persona.java

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

Ricordiamo che, 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 la fase di progetto ha stabilito se dobbiamo utilizzare nella realizzazione un approccio di verifica lato client o lato server.

Per completezza, vedremo ora il codice della classe `Persona` con verifica lato server. Successivamente, per pure esigenze di compattezza del codice mostrato, adotteremo l'approccio lato client.

Verifica nel lato server: esempio

```
// File SoloAttributi/VerificaLatoServer/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)";
  }
}
```


Esempio di cliente

Supponiamo che nella fase di analisi sia stata data la seguente specifica.

InizioSpecificaOperazioni Analisi Statisitica

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

FineSpecifica

Nella fase di progetto è stato specificato un algoritmo (omesso per brevità) ed è stato deciso di rappresentare l'input dell'operazione mediante la classe Java `Set`.

In questa parte del corso la specifica delle operazioni mediante notazione formale viene sempre lasciata come esercizio.

Realizzazione del cliente

```
// File SoloAttributi/AnalisiStatistica.java

import java.util.*;

public final class AnalisiStatistica {
    public static int quantiConiugati(Set<Persona> i) {
        int quanti = 0;
        Iterator<Persona> it = i.iterator();
        while(it.hasNext()) {
            Persona elem = it.next();
            if (elem.getConiugato())
                quanti++;
        }
        return quanti;
    }
    private AnalisiStatistica() {}
}
```


Molteplicità di attributi

Quando la classe UML *C* ha attributi UML con una loro molteplicità (ad es., *numTel: stringa* $\{0..*\}$), possiamo usare per la loro rappresentazione una classe contenitore apposita, come `HashSet<String>`.

In particolare, va previsto un campo dato di tale classe, che va inizializzato con `new()` dal costruttore della classe Java `C`.

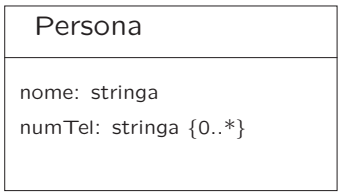
Per la gestione di questo campo vanno previste opportune funzioni `public`:

- per la scrittura del campo sono necessarie due funzioni, rispettivamente per l'inserimento di elementi nell'insieme e per la loro cancellazione;
- per la lettura del campo è necessaria una funzione **get**.

Molteplicità di attributi: esempio

Realizziamo la classe *Persona* in maniera che ogni persona possa avere un numero qualsiasi di numeri di telefono.

Facciamo riferimento alla seguente classe UML.



Realizzazione in Java

```
// File MolteplicitaAttributi/Persona.java
import java.util.*;
public class Persona {
    private final String nome;
    private HashSet<String> numTel;
    public Persona(String n) {
        numTel = new HashSet<String>();
        nome = n;
    }
    public String getNome() {
        return nome;
    }
    public void aggiungiNumTel(String n) {
        if (n != null) numTel.add(n);
    }
    public void eliminaNumTel(String n) {
        numTel.remove(n);
    }
    public Set<String> getNumTel() {
        return (HashSet<String>)numTel.clone();
    }
    public String toString() {
        return nome + ' ' + numTel;
    }
}
```

Classe Java Persona: considerazioni

- La classe ha un campo dato di tipo `HashSet`.
- Il costruttore della classe `Persona` crea un oggetto di tale classe, usandone il costruttore. Di fatto, viene creato un insieme vuoto di riferimenti di tipo `String`.
- Ci sono varie funzioni che permettono di gestire l'insieme:
  - `aggiungiNumTel(String)`: permette di inserire un nuovo numero telefonico; il fatto che non vengano creati duplicati nella struttura di dati è garantito dal funzionamento della funzione `Set.add()`, che verifica tramite la funzione `equals()` (in questo caso di `String`) l'eventuale presenza dell'oggetto di cui si richiede l'inserimento;
  - `eliminaNumTel(Stringa)`: permette di eliminare un numero telefonico;
  - `getNumTel()`: permette di ottenere tutti i numeri telefonici di una persona.

Classe Java Persona: considerazioni (cont.)

- Si noti che la funzione `getNumTel()` restituisce un `Set<String>`. L'uso dell'interfaccia `Set` invece di una classe concreta che la realizza (come `HashSet`) permette ai clienti della classe di astrarre della specifica struttura dati utilizzata per realizzare le funzionalità previste da `Set`, aumentando così l'information hiding.

Classe Java Persona: considerazioni (cont.)

- Si noti che la funzione `getNumTel()` restituisce una **copia** dell'insieme dei numeri di telefono (ovvero della struttura di dati), in quanto abbiamo scelto che l'attributo *numTel* venga gestito solamente dalla classe `Persona`.
- Se così non fosse, daremmo al cliente della classe `Persona` la possibilità di modificare l'insieme che rappresenta l'attributo *numTel* a suo piacimento, distruggendo la modularizzazione.
- Queste considerazioni valgono ogni volta che restituiamo un valore di un tipo UML realizzato mediante una classe Java i cui *oggetti sono mutabili*.

Cliente della classe Java Persona

Per comprendere meglio questo aspetto, consideriamo un cliente della classe `Persona` specificato come segue.

InizioSpecificaOperazioni Gestione Rubrica

TuttiNumTel (*p1*: *Persona*, *p2*: *Persona*): *Insieme(stringa)*
pre: nessuna
post: *result* è l'insieme unione dei numeri di telefono di *p1* e *p2*

FineSpecifica

Cliente della classe Java Persona (cont.)

Per l'operazione *TuttiNumTel(p1,p2)* adottiamo il seguente algoritmo:

```
Insieme(stringa) result = p1.numTel;
per ogni elemento el di p2.numTel
    aggiungi el a result
return result
```


Cliente della classe Java Persona (cont.)

```
// File MolteplicitaAttributi/GestioneRubrica.java
import java.util.*;

public final class GestioneRubrica {
    public static Set<String> tuttiNumTel
        (Persona p1, Persona p2) {
        Set<String> result = p1.getNumTel();
        Iterator<String> it = p2.getNumTel().iterator();
        while(it.hasNext())
            result.add(it.next());
        return result;
    }
    private GestioneRubrica() { };
}
```

Questa funzione farebbe **side-effect indesiderato** su `p1` se `getNumTel()` non restituisse una copia dell'insieme dei numeri di telefono.

Considerazioni sul cliente

Notiamo che la funzione cliente `tuttiNumTel()` si basa sull'assunzione che la funzione `getNumTel()` **restituisca una copia** della struttura di dati che rappresenta i numeri di telefono.

Se così non fosse (cioè se la funzione `tuttiNumTel()` non lavorasse su una copia, ma sull'originale) verrebbe completamente distrutta la struttura di dati, mediante le ripetute operazioni di inserimento.

L'errore di progettazione che consiste nel permettere al cliente di distruggere le strutture di dati private di un oggetto si chiama *interferenza*.

Esercizio 1: altro cliente della classe

Realizzare in Java le seguenti operazioni *Analisi Recapiti*:

InizioSpecificaOperazioni Analisi Recapiti

Convivono (*p1*: *Persona*, *p2*: *Persona*): *booleano*
pre: nessuna
post: *result* vale *true* se *p1* e *p2* hanno almeno un numero telefonico in comune, vale *false*, altrimenti

FineSpecifica

Altra realizzazione della classe Java Persona

- La funzione `getNumTel()`, che permette di ottenere tutti i numeri telefonici di una persona, potrebbe essere realizzata restituendo un *iteratore* dell'insieme dei numeri di telefono.
- Il vantaggio di questa scelta consiste in un minore utilizzo di memoria.
- Lo svantaggio risiede nel fatto che tipicamente i clienti devono realizzare funzioni più complesse.
- Per eliminare la possibilità che i clienti facciano interferenza, restituiamo un iteratore realizzato tramite la classe `IteratoreSolaLettura<T>`, che elimina `remove()` da `Iterator`.

Schemi realizzativi

Riassumendo, possiamo scegliere di realizzare la classe `Persona` attraverso due *schemi realizzativi* differenti.

	`getNumTel()`	Vantaggi	Svantaggi
Senza condivisione memoria	restituisce copia profonda (`clone()`)	cliente più semplice	potenziale spreco memoria
Con condivisione memoria	restituisce `IteratoreSolaLettura`	cliente più complicato	risparmio memoria

Schema realizzativo con condivisione

```
// File MolteplicitaAttributiCond/Persona.java
import java.util.*;
import IteratoreSolaLettura.*;
public class Persona {
    private final String nome;
    private HashSet<String> numTel;
    public Persona(String n) {
        numTel = new HashSet<String>();
        nome = n;
    }
    public String getNome() { return nome; }
    public void aggiungiNumTel(String n) {
        if (n != null) numTel.add(n);
    }
    public void eliminaNumTel(String n) {
        numTel.remove(n);
    }
    public Iterator<String> getNumTel() {
        return new IteratoreSolaLettura<String>(numTel.iterator());
    }
    public String toString() {
        return nome + ' ' + numTel;
    }
}
```


La classe Java IteratoreSolaLettura

```
// File IteratoreSolaLettura/IteratoreSolaLettura.java
package IteratoreSolaLettura;
import java.util.*;
public class IteratoreSolaLettura<T> implements Iterator<T> {
    // elimina remove() da Iterator
    private Iterator<T> i;
    public IteratoreSolaLettura(Iterator<T> it) { i = it; }
    public T next() { return i.next(); }
    public boolean hasNext() { return i.hasNext(); }
    public void remove() {
        throw new UnsupportedOperationException();
    }
}
```

Esercizio 2: clienti per la nuova versione della classe Persona

Facendo riferimento all'ultima realizzazione della classe `Persona` (quella con lo schema realizzativo con condivisione di memoria), realizzare le operazioni *tuttiNumTel()* e *Convivono()* come opportune funzioni cliente.

Realizzazione di classe con attributi e operazio

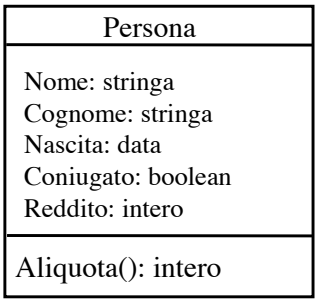
- Si procede come prima per quanto riguarda gli attributi.
- Si analizza la specifica della classe UML *C* e gli algoritmi associati alle operazioni di tale classe, che forniscono le informazioni sul significato di ogni operazione.
- Ogni operazione viene realizzata da una funzione `public` della classe Java.

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

Singola classe con attr. e operazioni: esempio

Consideriamo un raffinamento della classe UML *Persona* vista in uno degli esempi precedenti.

Si noti che ora una persona ha anche un reddito.



Specifica della classe UML

InizioSpecificaOperazioniClasse 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 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();
  }
}
```

Esercizio 3: classi UML con operazioni

Realizzare in Java la classe UML *Persona* che comprende anche l'operazione *Età*:

InizioSpecificaOperazioniClasse Persona

Aliquota (): *intero* ...

Età (*d*: *data*): *intero*
pre: *d* non è precedente a *this.Nascita*
post: *result* è l'età (in mesi compiuti) della persona *this* alla data *d*

FineSpecifica

Esercizio 4: cliente della classe

Realizzare in Java il seguente cliente *Analisi Redditi*:

InizioSpecificaOperazioni Analisi Redditi

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 nell'insieme di persone *i*

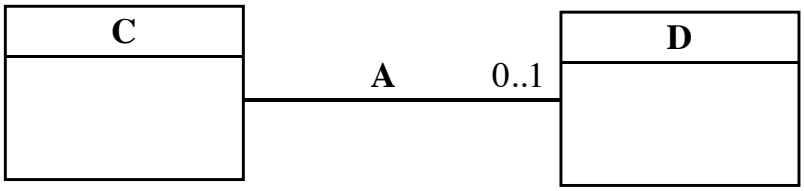
FineSpecifica

Realizzazione di associazioni

Nell'esposizione di questo argomento, seguiremo quest'ordine:

- associazioni binarie, con molteplicità 0..1, a responsabilità singola, senza attributi;
- associazioni binarie, con molteplicità 0..*, a responsabilità singola, senza attributi;
- associazioni binarie, con molteplicità 0..1, a responsabilità singola, con attributi;
- associazioni binarie a responsabilità doppia;
- associazioni binarie, con molteplicità diversa da 0..1 e 0..*;
- associazioni n-arie;
- associazioni ordinate.

Associazione con molteplicità 0..1 a responsabilità singola, senza attributi



Consideriamo il caso in cui

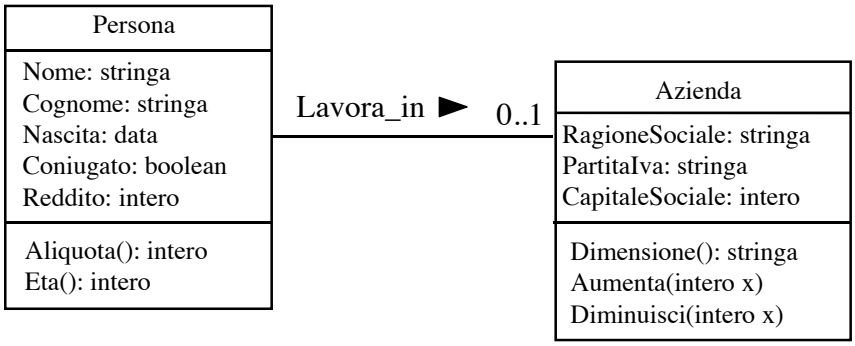
- l'associazione sia binaria;
- l'associazione colleghi ogni istanza di *C* a zero o una istanza di *D* (molteplicità 0..1),
- la *tabella delle responsabilità* prodotta in fase di progetto ci dica che *C* è l'unica ad avere responsabilità sull'associazione *A* (cioè dobbiamo realizzare un "solo verso" della associazione)
- l'associazione *A* non abbia attributi.

Associazione con molteplicità 0..1 a responsabilità singola, senza attributi (cont.)

In questo caso, la realizzazione 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 la 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



Assumiamo di avere stabilito, nella fase di progetto, che:

- la ragione sociale e la partita Iva di un'azienda **non cambiano**;
- 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).

Specifica della classe UML Azienda

InizioSpecificaOperazioniClasse 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

Classe Java Azienda

```
// File 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();
  }
}
```

Classe Java 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 5: cliente

Realizzare in Java il cliente *Analisi Aziende*, specificato di seguito:

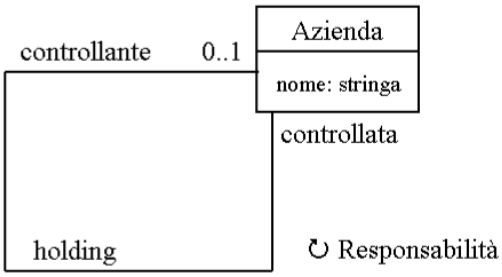
InizioSpecificaOperazioni Analisi Aziende

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

Associazioni che insistono più volte sulla stessa classe

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 alle classi.



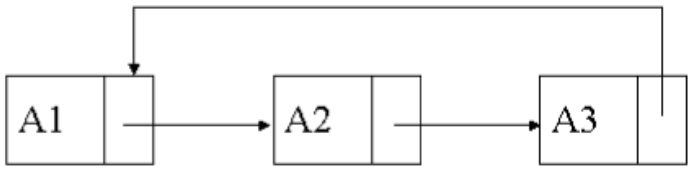
Supponiamo che la classe *Azienda* abbia la responsabilità su *holding*, solo nel ruolo *controllata*. Questo significa che, dato un oggetto *x* della classe *Azienda*, vogliamo poter eseguire operazioni su *x* per conoscere l'azienda controllante, per aggiornare l'azienda controllante, ecc.

Associazioni che insistono più volte sulla stessa classe: esempio

In questo caso, il nome del campo dato che rappresenta l'associazione viene in genere scelto uguale al nome del ruolo (nell'esempio, il nome è `controllante`).

```
// File Ruoli/Azienda.java

public class Azienda {
    private final String nome;
     private Azienda controllante; // il nome del campo è uguale al ruolo
    public Azienda(String n) { nome = n; }
    public Azienda getControllante() { return controllante; }
    public void setControllante(Azienda a) { controllante = a; }
     public String toString() {
        return nome + ((controllante == null)?"":
                       (" controllata da: "+controllante));
    }
}
```

Potenziale situazione anomala



L'azienda A1 ha come controllante A2, che ha come controllante A3, che ha a sua volta come controllante A1.

Diciamo che L'azienda A1 è "di fatto controllata da se stessa".

Esercizio 6: cliente

Realizzare in Java il cliente *Ricognizione truffe*, specificato di seguito:

InizioSpecificaOperazioni Ricognizione truffe

ControllataDaSeStessa (*a: Azienda): booleano*
pre: nessuna
post: *result* vale true se *a* ha se stessa come controllante o se, ciò è vero (ricorsivamente) per la sua controllante.

FineSpecifica

Associazioni con molteplicità 0..* a responsabilità singola, senza attributi

Ci concentriamo su associazioni binarie **con molteplicità 0..***, con le seguenti assunzioni:

- non abbiano attributi di associazione;
- solo una delle due classi *ha responsabilità* sull'associazione (dobbiamo rappresentare **un solo verso** dell'associazione).

Gli altri casi verranno considerati in seguito.

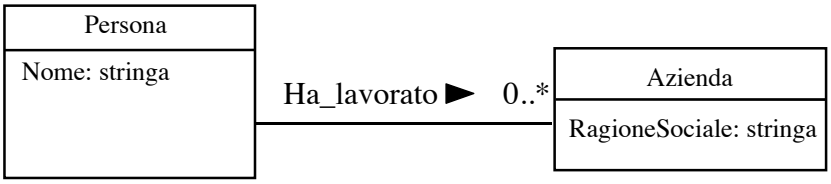
Associazioni con molteplicità 0..* a responsabilità singola, senza attributi (cont.)

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 dato di un tipo opportuno (ad esempio `HashSet`), per rappresentare la struttura di dati;
- dei campi funzione che permettano di gestire tale struttura di dati (funzioni `get`, `inserisci`, `elimina`).

Associazioni con molteplicità 0..* a responsabilità singola, senza attributi: esemp



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

Assumiamo anche che dai requisiti si evinca che è possibile eliminare un link di tipo *Ha_lavorato*.

Classe Java Persona

```
// File AssOSTAR/Persona.java
import java.util.*;
public class Persona {
  private final String nome;
  private HashSet<Azienda> insieme_link;
  public Persona(String n) {
    nome = n;
    insieme_link = new HashSet<Azienda>();
  }
  public String getNome() { return nome; }
  public void inserisciLinkHaLavorato(Azienda az) {
    if (az != null) insieme_link.add(az);
  }
  public void eliminaLinkHaLavorato(Azienda az) {
    if (az != null) insieme_link.remove(az);
  }
  public Set<Azienda> getLinkHaLavorato() {
    return (HashSet<Azienda>)insieme_link.clone();
  }
}
```

Classe Java Persona: considerazioni

- La classe ha un campo dato di tipo `HashSet<Azienda>`.
- Il costruttore della classe `Persona` crea un oggetto di tale classe, usandone il costruttore. Di fatto, viene creato un insieme vuoto di riferimenti di tipo `Azienda`.
- Ci sono varie funzioni che permettono di gestire l'insieme:
  - `inserisciLinkHaLavorato(Azienda)`: permette di inserire un nuovo link;
  - `eliminaLinkHaLavorato(Azienda)`: permette di eliminare un link esistente;
  - `getLinkHaLavorato()`: permette di ottenere tutti i link di una persona.

Classe Java Persona: considerazioni (cont.)

- Si noti che la funzione `getLinkHaLavorato()` restituisce un `Set<Azienda>` e non un `HashSet<Azienda>`. Come detto precedentemente nel caso di attributi con molteplicità 0..*, l'uso dell'interfaccia `Set` invece di una classe concreta che la realizza (come `HashSet`) permette ai clienti della classe di astrarre della specifica struttura dati utilizzata per realizzare le funzionalità previste da `Set`, aumentando così l'information hiding.

Classe Java Persona: considerazioni (cont.)

- Seguendo lo schema realizzativo senza condivisione di memoria, la funzione `getLinkHaLavorato()` restituisce una **copia** dell'insieme dei link (ovvero della struttura di dati).
- Questa situazione è infatti analoga a quella degli attributi di classe con molteplicità 0..* visti in precedenza, e scegliamo che i link dell'associazione *Ha-Lavorato* vengano gestiti solamente dalla classe `Persona`, che ha responsabilità sull'associazione.
- Per semplicità, nel seguito utilizzeremo sempre lo schema realizzativo senza condivisione di memoria.

Esercizio 7: cliente

Realizzare in Java il cliente *Analisi del collocamento*, specificato di seguito:

InizioSpecificaOperazioni Analisi del collocamento

PiùDipendenti *(i: Insieme(Persona)): Azienda*
pre: nessuna
post: *result* è l'azienda per cui più dipendenti fra le persone di *i* hanno lavorato

FineSpecifica

Esercizio 8: schema realizzativo con condivisione

Realizzare in Java la classe `Persona` mediante lo schema realizzativo con condivisione di memoria e il cliente *Analisi del collocamento* che fa uso di tale versione della classe.

Attributi di associazione

Consideriamo il caso in cui la classe *C* sia l'unica ad avere la responsabilità sull'associazione *A*, e l'associazione *A* abbia uno o più **attributi** di molteplicità 1..1.

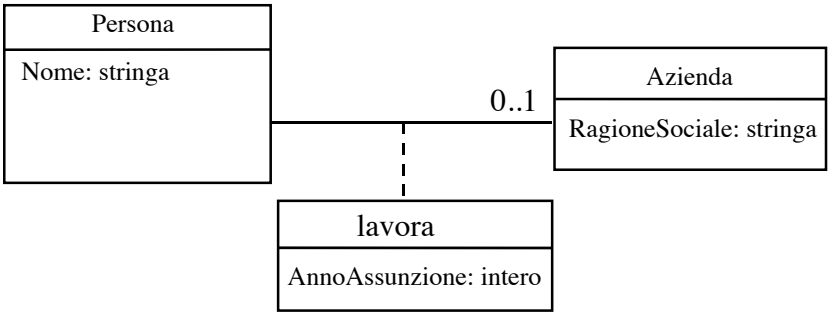
Considereremo

- inizialmente che A abbia, rispetto a C, molteplicità 0..1;
- dopo anche molteplicità 0..*.

Gli altri casi, come altre molteplicità per attributi *(immediato)*, o responsabilità sull'associazione di entrambe le classi *(difficile)*, verranno considerati in seguito.

Rappresentazione di attributi di associazione realizzazione naive

Esempio (solo *Persona* ha responsabilità sull'associazione):



Realizzazione naive:

1. si aggiunge alla classe *C* un campo per ogni attributo dell'associazione *A*, che viene trattato in modo simile ad un attributo della classe *C*.
2. si fa uso di una struttura di dati ad hoc per rappresentare istanze dell'associazione (link).

Rappresentazione di attributi di associazione realizzazione naive (cont.)

Consideriamo l'esempio, sceglendo la **prima** strategia.

```
// File Ass01Attr-NoLink/Persona.java

public class Persona {
  private final String nome;
  private Azienda lavora;
  private int annoAssunzione;
  public Persona(String n) { nome = n; }
  public String getNome() { return nome; }
  public Azienda getLavora() { return lavora; }
  public int getAnnoAssunzione() throws EccezionePrecondizioni {
if (lavora==null)
throw new EccezionePrecondizioni(this + " Non partecipa alla associazione Lavora");
return annoAssunzione;
  }
  public void setLavora(Azienda a, int x) {
       if (a != null) { lavora = a; annoAssunzione = x; }
  }
  public void eliminaLavora() { lavora = null; }
}
```

Classe Java EccezionePrecondizioni

```
// File Ass01Attr/EccezionePrecondizioni.java
public class EccezionePrecondizioni extends Exception {
  private String messaggio;
  public EccezionePrecondizioni(String m) {
    messaggio = m;
  }
  public EccezionePrecondizioni() {
    messaggio = "Si e' verificata una violazione delle precondizioni";
  }
  public String toString() {
    return messaggio;
  }
}
```


Rappresentazione di attributi di associazione realizzazione naive: osservazioni

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 molteplicità `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`). Altrimenti viene generata una opportuna istanza di `EccezionePrecondizioni`.

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.

Importante: questa strategia realizzativa naive non può essere estesa ad associazioni con molteplicità 0..!*

Attributi di associazione: realizzazione

Consideriamo adesso una strategia di realizzazione più ragionata, che è quella da preferirsi.

La presenza degli attributi sull'associazione impedisce di usare i meccanismi base di Java (cioè i riferimenti) per rappresentare i link UML.

Dobbiamo quindi rappresentare la nozione di link in modo esplicito attraverso una classe.

Attributi di associazione: realizzazione (cont.)

Per rappresentare l'associazione *A* fra le classi UML *C* e *D* introduciamo **una ulteriore classe** Java `TipoLinkA`, che ha lo scopo di rappresentare i link (tuple -in questo caso coppie) fra gli oggetti delle classi *C* e *D*.

Si noti che questi link (tuple) sono **valori**, non oggetti. Quindi la classe `TipoLinkA` rappresenta un **tipo**, non una classe UML.

In particolare, ci sarà un oggetto di classe `TipoLinkA` per ogni link (presente al livello estensionale) fra un oggetto di classe *C* ed uno di classe *D*.

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

- gli attributi dell'associazione;
- i riferimenti agli oggetti delle classi `C` e `D` che costituiscono le componenti della tupla che il link rappresenta *(Nota per essere precisi tali riferimenti sono variabili che contengano gli* **identificatori** *degli oggetti coinvolti)*.

Funzioni della classe Java TipoLinkA

La classe Java `TipoLinkA` avrà inoltre le seguenti funzioni:

- funzioni per la gestione dei suoi campi dati:
  - costruttore (lancia un'eccezione di tipo `EccezionePrecondizioni` se i riferimenti di tipo `C` e `D` passati come argomenti sono `null`),
  - funzioni **get**;
- funzione `equals()` ridefinita in maniera tale da verificare l'uguaglianza solo sugli oggetti collegati dal link, **ignorando gli attributi**.
- funzione `hashCode()` ridefinita in maniera tale da verificare il principio secondo il quale se due oggetti sono uguali secondo `equals()` allora questi devono avere lo stesso codice di hash secondo `hashCode()`.

Non avrà invece funzioni **set**: i suoi oggetti sono *immutabili*, ovvero una volta creati non possono più essere cambiati.

Attributi di associazione (cont.)

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

In particolare, la classe Java `C` avrà:

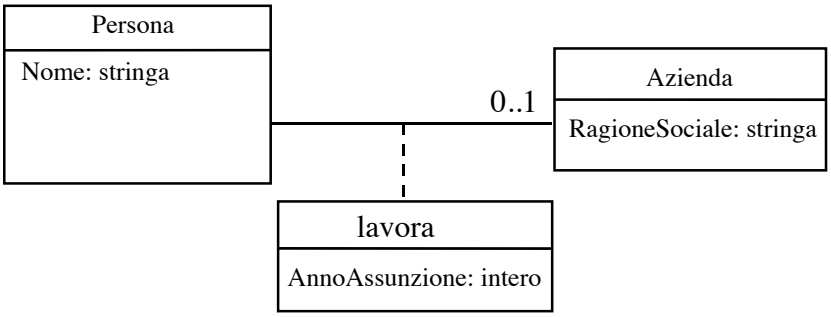
- un campo dato di tipo `TipoLinkA`, per rappresentare l'eventuale link;

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

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

Attributi di associazione: realizzazione

Mostriamo nel dettaglio la realizzazione proposta sull'esempio già visto:



Ricordiamo che stiamo assumendo 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).

Classe Java TipoLinkLavora

```
// File Ass01Attr/TipoLinkLavora

public class TipoLinkLavora {
  private final Persona laPersona;
  private final Azienda laAzienda;
  private final int annoAssunzione;
  public TipoLinkLavora(Azienda x, Persona y, int a)
    throws EccezionePrecondizioni {
    if (x == null || y == null)  // CONTROLLO PRECONDIZIONI
      throw new EccezionePrecondizioni
        ("Gli oggetti devono essere inizializzati");
    laAzienda = x; laPersona = y; annoAssunzione = a;
  }
  public boolean equals(Object o) {
    if (o != null && getClass().equals(o.getClass())) {
      TipoLinkLavora b = (TipoLinkLavora)o;
      return b.laPersona == laPersona && b.laAzienda == laAzienda;
    }
    else return false;
  }
  public int hashCode() {
    return laPersona.hashCode() + laAzienda.hashCode();
  }
  public Azienda getAzienda() { return laAzienda; }
```

```
  public Persona getPersona() { return laPersona; }
  public int getAnnoAssunzione() { return annoAssunzione; }
}
```

Classe Java Persona

```
// File 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.getPersona() == this)
            link = t;
    }
    public void eliminaLinkLavora() {
        link = null;
    }
    public TipoLinkLavora getLinkLavora() { return link;  }
}
```

Considerazioni sulle classi Java

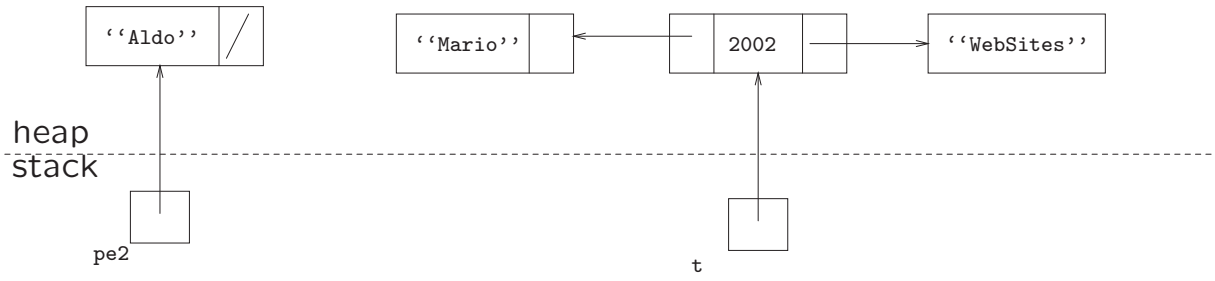
- Si noti che i campi dati nella classe `TipoLinkLavora` sono tutti `final`.

  Di fatto un oggetto di tale 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 sia già associata ad un link;
  - l'oggetto che rappresenta il link esista;
  - la persona a cui si riferisce il link sia l'oggetto di invocazione.

- Per **cambiare** l'oggetto della classe `Azienda` a cui una persona è legata tramite l'associazione `lavora` è necessario invocare prima `eliminaLinkLavora()` e poi `inserisciLinkLavora()`.

Controllo coerenza riferimenti

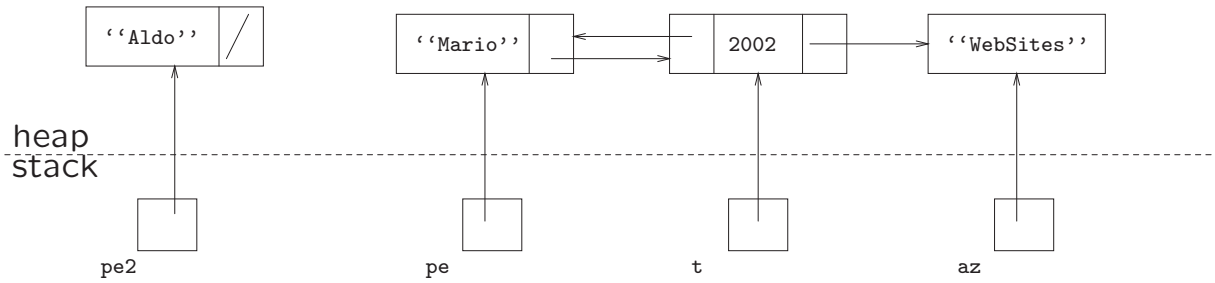


Il link `t` non si riferisce all'oggetto "Aldo".

Quindi, se chiediamo all'oggetto "Aldo" di inserire tale link, non deve essere modificato nulla.

Infatti la funzione `inserisciLinkLavora()` della classe `Persona` si assicura che 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

```
Azienda az = new Azienda("WebSites");
Persona pe = new Persona("Mario"),
    pe2 = new Persona("Aldo");
TipoLinkLavora t = null;
try {
    t = new TipoLinkLavora(az,pe,2002);
}
catch (EccezionePrecondizioni e) {
    System.out.println(e);
}
pe.inserisciLinkLavora(t);
```

Esercizio 9: cliente

Realizzare in Java il cliente *Ristrutturazione Industriale*, specificato di seguito:

InizioSpecificaOperazioni 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*

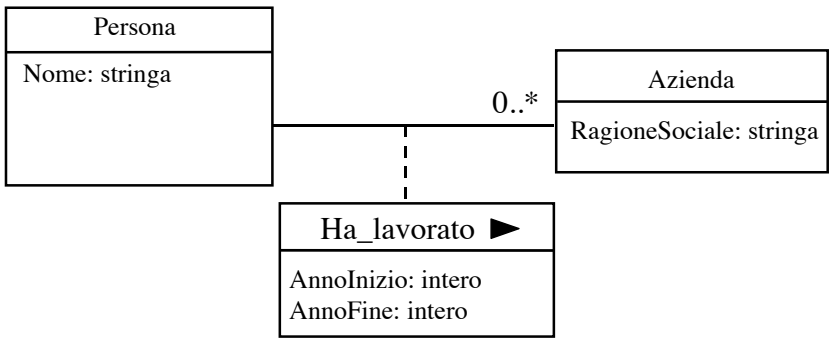
AssunzionePersonaleEsperto *(i: Insieme(Persona), a: Azienda, av: intero, an: intero)*
pre: $an \geq av$
post: tutte le persone nell'insieme di persone *i* che lavorano in un'azienda qualsiasi fin dall'anno *av* vengono assunte dall'azienda *a* nell'anno *an*

FineSpecifica

Associazioni 0..* con attributi

Ci concentriamo ora su associazioni binarie con molteplicità 0..*, e **con attributi**. Ci riferiremo al seguente esempio (si noti che non è possibile rappresentare che una persona ha lavorato due o più volte per la stessa azienda). Assumiamo per semplicità che si lavori sempre per anni interi.

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



Associazioni 0..* con attributi (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`);

   inoltre, dobbiamo prevedere la possibilità che il costruttore di questa classe lanci un'eccezione nel caso in cui i riferimenti passatigli come argomento siano pari a `null`;

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

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** e la funzione `hashCode` ridefinita di conseguenza.

Rappresentazione dei link in Java

```
// File AssOSTARAttr/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)
        throws EccezionePrecondizioni {
        if (x == null || y == null)  // CONTROLLO PRECONDIZIONI
            throw new EccezionePrecondizioni
                ("Gli oggetti devono essere inizializzati");
        laAzienda = x; laPersona = y; annoInizio = ai; annoFine = af;
    }
    public boolean equals(Object o) {
        if (o != null && getClass().equals(o.getClass())) {
            TipoLinkHaLavorato b = (TipoLinkHaLavorato)o;
            return b.laPersona == laPersona && b.laAzienda == laAzienda;
        }
        else return false;
    }
    public int hashCode() {
        return laPersona.hashCode() + laAzienda.hashCode();
    }
    public Azienda getAzienda() { return laAzienda; }
```



```
    public Persona getPersona() { return laPersona; }
    public int getAnnoInizio() { return annoInizio; }
    public int getAnnoFine() { return annoFine; }
}
```

Classe Java 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 la classe Java `Set`.

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

Classe Java Persona

```
// File AssOSTARAttr/Persona.java
import java.util.*;
public class Persona {
    private final String nome;
    private HashSet<TipoLinkHaLavorato> insieme_link;
    public Persona(String n) {
        nome = n;
        insieme_link = new HashSet<TipoLinkHaLavorato>();
    }
    public String getNome() { return nome; }
    public void inserisciLinkHaLavorato(TipoLinkHaLavorato t) {
        if (t != null && t.getPersona() == this)
            insieme_link.add(t);
    }
    public void eliminaLinkHaLavorato(TipoLinkHaLavorato t) {
        if (t != null && t.getPersona() == this)
            insieme_link.remove(t);
    }
    public Set<TipoLinkHaLavorato> getLinkHaLavorato() {
        return (HashSet<TipoLinkHaLavorato>)insieme_link.clone();
    }
}
```

Riassunto metodi per responsabilità singola

La seguente tabella riassume gli argomenti ed i controlli necessari per le funzioni di inserimento e cancellazione nei casi di associazione (a responsabilità singola) finora esaminati.

		0..1		0..*	
		no attr.	attributo	no attr.	attributo
inser.	arg.	rif. a oggetto	rif. a link	rif. a oggetto	rif. a link
	controllo	–	arg `!= null` link si rif. a `this` link `== null`	arg `!= null`	arg `!= null` link si rif. a `this`
canc.	arg.	`null`	nessuno	rif. a oggetto	rif. a link
	controllo	–	–	arg `!= null`	arg `!= null` link si rif. a `this`

Esercizio 10: cliente

Realizzare in Java il cliente *Analisi Mercato Lavoro*:

InizioSpecificaOperazioni 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

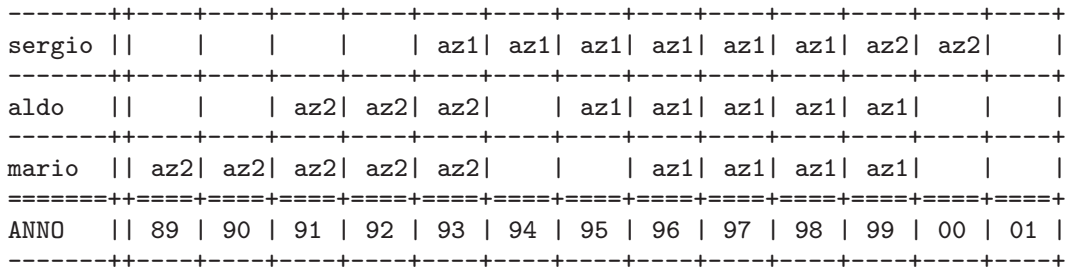
RiAssuntoSubito *(p: Persona): booleano*
pre: nessuna
post: *result* vale *true* se e solo se *p* ha lavorato consecutivamente per due aziende (anno di inizio per un'azienda uguale all'anno di fine per un'altra azienda + 1)

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

Esercizio 11

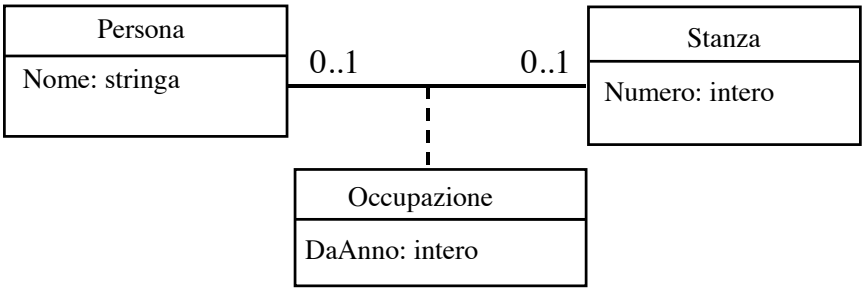
Verificare le funzioni realizzate facendo riferimento al seguente caso di test.



Responsabilità di entrambe le classi UML

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.

Resp. di entrambe le classi UML: esempio



Supponiamo che sia *Persona* sia *Stanza* abbiano responsabilità sull'associazione.

Resp. di entrambe le classi UML (cont.)

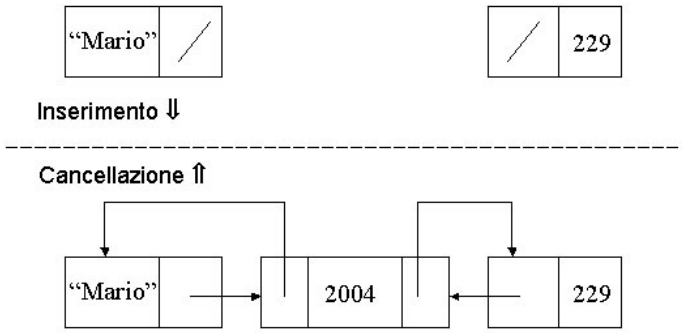
Problema di fondo:
quando creiamo un link fra un oggetto Java `pe` di classe `Persona` un oggetto Java `st` di classe `Stanza`, dobbiamo cambiare lo stato **sia di** `pe` **sia di** `st`.

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.

Mantenimento coerenza



Resp. di entrambe le classi UML (cont.)

Chiaramente, non possiamo dare al cliente delle classi `Persona` e `Stanza` questo onere, che deve essere gestito invece da queste ultime.

Per motivi che saranno chiariti in seguito, è preferibile **centralizzare** la responsabilità di assegnare i riferimenti in maniera corretta.

In particolare, realizziamo una ulteriore classe Java (chiamata `ManagerOccupazione`) 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 utilizzare (come in tutti i casi in cui c'è necessità di rappresentare attributi di associazione) una classe Java per i link, in questo caso `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`, che viene inizializzato a `null` dal costruttore;
- funzioni per la gestione di questo campo, in particolare:
  - `void inserisciLinkOccupazione(TipoLinkOccupazione)`, per associare un link all'oggetto, ma che delega l'operazione effettiva a `ManagerOccupazione`;
  - `void eliminaLinkOccupazione(TipoLinkeOccupazione)`, per rimuovere l'associazione di un link all'oggetto, ma che anche esso delega l'operazione effettiva `ManagerOccupazione`;
  - `TipoLinkOccupazione getLinkOccupazione()`, per interrogare l'oggetto;

  Funzioni speciali utilizzabili solo da `ManagerOccupazione`:
  - `void inserisciPerManagerOccupazione(ManagerOccupazione)`, funzione speciale per gestire l'inserimento di link `TipoLinkeOccupazione` nelle proprie strutture dati (molto semplici in questo caso);
  - `void eliminaPerManagerOccupazione(ManagerOccupazione)`, funzione speciale per gestire eliminazione di link `TipoLinkeOccupazione` nelle proprie strutture dati.

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

Classe Java Persona

```
// File RespEntrambi01/Persona.java
public class Persona {
  private final String nome;
  private TipoLinkOccupazione link;
  public Persona(String n) { nome = n; }
  public String getNome() { return nome; }
  public void inserisciLinkOccupazione(TipoLinkOccupazione t) {
      if (t != null && t.getPersona()==this)
         ManagerOccupazione.inserisci(t);
  }
  public void eliminaLinkOccupazione(TipoLinkOccupazione t) {
      if (t != null && t.getPersona()==this)
         ManagerOccupazione.elimina(t);
  }
  public TipoLinkOccupazione getLinkOccupazione() {
      return link;
  }
  public void inserisciPerManagerOccupazione(ManagerOccupazione a) {
      if (a != null) link = a.getLink();
  }
  public void eliminaPerManagerOccupazione(ManagerOccupazione a) {
      if (a != null) link = null;
  }
}
```

Classe Java Stanza

```
// File RespEntrambi01/Stanza.java
public class Stanza {
  private final int numero;
  private TipoLinkOccupazione link;
  public Stanza(int n) { numero = n; }
  public int getNumero() { return numero; }
  public void inserisciLinkOccupazione(TipoLinkOccupazione t) {
      if (t != null && t.getStanza()==this)
         ManagerOccupazione.inserisci(t);
  }
  public void eliminaLinkOccupazione(TipoLinkOccupazione t) {
      if (t != null && t.getStanza()==this)
         ManagerOccupazione.elimina(t);
  }
  public TipoLinkOccupazione getLinkOccupazione() {
      return link;
  }
  public void inserisciPerManagerOccupazione(ManagerOccupazione a) {
      if (a != null) link = a.getLink();
  }
  public void eliminaPerManagerOccupazione(ManagerOccupazione a) {
      if (a != null) link = null;
  }
}
```

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; lancia un'eccezione se i riferimenti alla stanza o alla persona sono `null`;
- tre funzioni `get`, per interrogare l'oggetto;
- la funzione `equals` per verificare l'uguaglianza solo sugli oggetti collegati dal link, ignorando gli attributi;
- la funzione `hashCode`, ridefinita di conseguenza.

Classe Java TipoLinkOccupazione

```
// File 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)
        throws EccezionePrecondizioni {
        if (x == null || y == null)  // CONTROLLO PRECONDIZIONI
            throw new EccezionePrecondizioni
                ("Gli oggetti devono essere inizializzati");
        laStanza = x; laPersona = y; daAnno = a;
    }
    public boolean equals(Object o) {
        if (o != null && getClass().equals(o.getClass())) {
            TipoLinkOccupazione b = (TipoLinkOccupazione)o;
            return b.laPersona == laPersona && b.laStanza == laStanza;
        }
        else return false;
    }
    public int hashCode() {
        return laPersona.hashCode() + laStanza.hashCode();
    }
    public Stanza getStanza() { return laStanza; }
```

```
    public Persona getPersona() { return laPersona; }
    public int getDaAnno() { return daAnno; }
}
```

Caratteristiche delle classi Java (cont.)

`ManagerOccupazione`: 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 associare 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 sarà **final**, per evitare che si possa definire una sottoclasse in cui il costruttore è pubblico.

Classe Java ManagerOccupazione

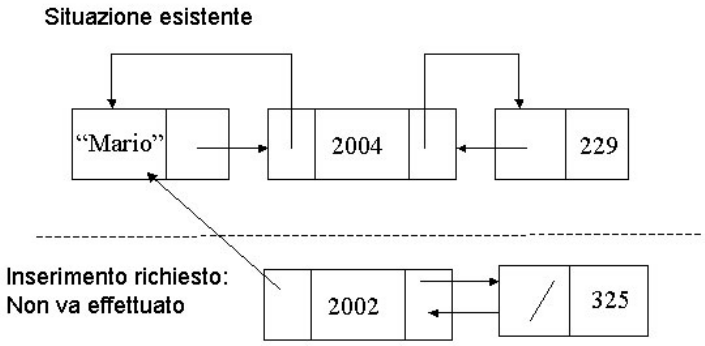
```
// File RespEntrambi01/ManagerOccupazione.java

public final class ManagerOccupazione {
    private ManagerOccupazione(TipoLinkOccupazione x) { link = x; }
    private TipoLinkOccupazione link;
    public TipoLinkOccupazione getLink() { return link; }
    public static void inserisci(TipoLinkOccupazione y) {
        if (y != null &&
            y.getPersona().getLinkOccupazione() == null &&
            y.getStanza().getLinkOccupazione() == null) {
            ManagerOccupazione k = new ManagerOccupazione(y);
            y.getStanza().inserisciPerManagerOccupazione(k);
            y.getPersona().inserisciPerManagerOccupazione(k);
        }
    }
    public static void elimina(TipoLinkOccupazione y) {
        if (y != null && y.getPersona().getLinkOccupazione().equals(y)) {
            ManagerOccupazione k = new ManagerOccupazione(y);
            y.getStanza().eliminaPerManagerOccupazione(k);
            y.getPersona().eliminaPerManagerOccupazione(k);
        }
    }
}
```


Inserimento di link: controlli

Si noti che è necessario prevenire la possibilità di richiedere agli oggetti di tipo `Stanza` o `Persona` di inserire link quando gli oggetti sono già "occupati" .

Per tale motivo la funzione `inserisci()` verifica (tramite `getPerManagerOccupazione()`) che il link `y` che le viene passato come argomento si riferisca ad oggetti di tipo `Stanza` e `Persona` che non sono associati ad alcun link di tipo *Occupazione*.

Inserimento di link: esempio

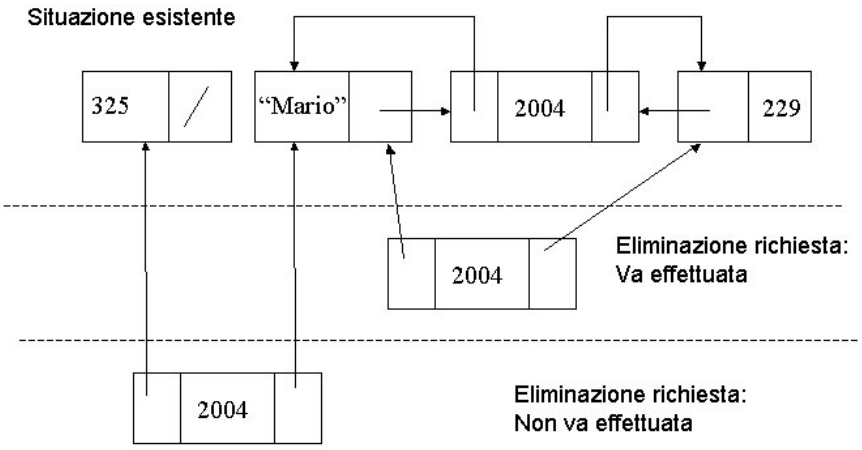


Eliminazione di link: controlli

Si noti che, al fine di prevenire la possibilità di richiedere agli oggetti di tipo `Stanza` o `Persona` di eliminare link inesistenti (creando in questa maniera situazioni inconsistenti) la funzione `elimina()` deve verificare (tramite `equals()`) che il link `y` che le viene passato come argomento si riferisca agli stessi oggetti di tipo `Stanza` e `Persona` del campo dato `link`.

Per fare ciò è sufficiente effettuare la verifica mediante il link da cui si arriva tramite la persona.

Eliminazione di link: esempio



Classe Java ManagerOccupazione (cont.)

Il costruttore della classe `ManagerOccupazione` 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 `ManagerOccupazione.inserisci()`;
- rimuovere link, mediante una chiamata alla funzione `ManagerOccupazione.elimina()`.

Si noti viene effettuato il controllo che gli argomenti di queste ultime due funzioni corrispondano ad oggetti (non siano `null`).

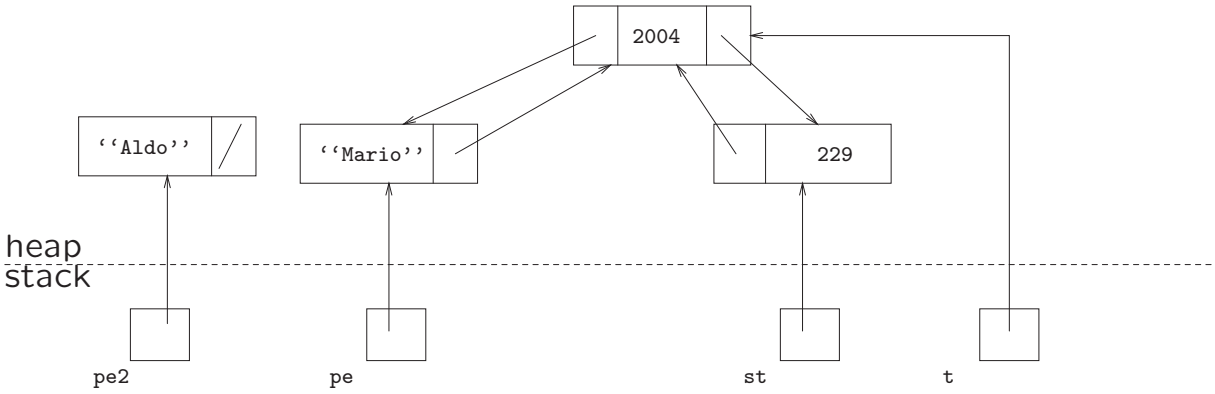
Considerazioni sulle classi Java

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

Considerazioni sulle classi Java(cont.)

- 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 `ManagerOccupazione` ...
- ... anche quando effettuano inserimenti e cancellazioni di link mediante le classi `Persona` e `Stanza`: queste infatti delegano tali inserimenti e cancellazioni alla classe `ManagerOccupazione`.

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 `ManagerOccupazione` non è direttamente accessibile dai clienti.

Realizzazione della situazione di esempio

```
Stanza st = new Stanza(229);

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

TipoLinkOccupazione t = null;
try {
    t = new TipoLinkOccupazione(st,pe,2004);
}
catch (EccezionePrecondizioni e) {
    System.out.println(e);
}
ManagerOccupazione.inserisci(t);
```

Esercizio 12: cliente

Realizzare in Java il cliente *Riallocazione Personale*:

InizioSpecificaOperazioni Riallocazione Personale

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

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

. . .

Esercizio 13: cliente

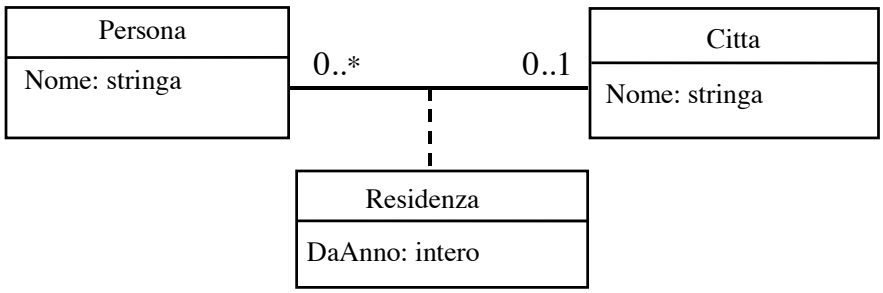
. . .

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

Resp. di entrambe le classi UML: 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 le seguenti:

- 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 `Set` viene creato tramite il costruttore;
  - ha tre campi funzione (`inserisciLinkResidenza()`, `eliminaLinkResidenza()` e `getLinkResidenza()`) per la gestione dell'insieme dei link; quest'ultima restituisce **una copia** dell'insieme dei link;
  - ha i due campi funzioni speciali di ausilio per `ManagerResidenza`.

Classe Java Citta

```
// File RespEntrambiOSTAR/Citta.java
import java.util.*;
public class Citta {
    private final String nome;
    private HashSet<TipoLinkResidenza> insieme_link;
    public Citta(String n) {
        nome = n;
        insieme_link = new HashSet<TipoLinkResidenza>();
    }
    public String getNome() { return nome; }
    public void inserisciLinkResidenza(TipoLinkResidenza t) {
      if (t != null && t.getCitta()==this)
         ManagerResidenza.inserisci(t);
    }
    public void eliminaLinkResidenza(TipoLinkResidenza t) {
      if (t != null && t.getCitta()==this)
         ManagerResidenza.elimina(t);
    }
    public Set<TipoLinkResidenza> getLinkResidenza() {
        return (HashSet<TipoLinkResidenza>)insieme_link.clone();
    }
    public void inserisciPerManagerResidenza(ManagerResidenza a) {
        if (a != null) insieme_link.add(a.getLink());
    }
```



```
    public void eliminaPerManagerResidenza(ManagerResidenza a) {
        if (a != null) insieme_link.remove(a.getLink());
    }
}
```

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

- La classe Java (nel nostro esempio: `Persona`) i cui oggetti possono essere legati al più ad un oggetto dell'altra classe Java (nel nostro esempio: `Citta`) è **esattamente identica** al caso di entrambe le 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.

Per completezza, viene riportato di seguito il codice di tutte le classi.

Classe Java Persona

```
// File RespEntrambi0STAR/Persona.java

public class Persona {
  private final String nome;
  private TipoLinkResidenza link;
  public Persona(String n) { nome = n; }
  public String getNome() { return nome; }
  public void inserisciLinkResidenza(TipoLinkResidenza t) {
      if (t != null && t.getPersona()==this)
         ManagerResidenza.inserisci(t);
  }
    public void eliminaLinkResidenza(TipoLinkResidenza t) {
      if (t != null && t.getPersona()==this)
         ManagerResidenza.elimina(t);
  }
  public TipoLinkResidenza getLinkResidenza() {
    return link;
  }
  public void inserisciPerManagerResidenza(ManagerResidenza a) {
      if (a != null) link = a.getLink();
  }
  public void eliminaPerManagerResidenza(ManagerResidenza a) {
      if (a != null) link = null;
  }
}
```

Classe Java TipoLinkResidenza

```
// File 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)
        throws EccezionePrecondizioni {
        if (x == null || y == null)  // CONTROLLO PRECONDIZIONI
            throw new EccezionePrecondizioni
                ("Gli oggetti devono essere inizializzati");
        laCitta = x; laPersona = y; daAnno = a;
    }
    public boolean equals(Object o) {
        if (o != null && getClass().equals(o.getClass())) {
            TipoLinkResidenza b = (TipoLinkResidenza)o;
            return b.laPersona == laPersona && b.laCitta == laCitta;
        }
        else return false;
    }
    public int hashCode() {
        return laPersona.hashCode() + laCitta.hashCode();
    }
    public Citta getCitta() { return laCitta; }
```



```
    public Persona getPersona() { return laPersona; }
    public int getDaAnno() { return daAnno; }
}
```

Classe Java ManagerResidenza

```
// File RespEntrambi0STAR/ManagerResidenza.java

public final class ManagerResidenza {
    private ManagerResidenza(TipoLinkResidenza x) { link = x; }
    private TipoLinkResidenza link;
    public TipoLinkResidenza getLink() { return link; }
    public static void inserisci(TipoLinkResidenza y) {
        if (y != null && y.getPersona().getLinkResidenza() == null) {
            ManagerResidenza k = new ManagerResidenza(y);
            y.getCitta().inserisciPerManagerResidenza(k);
            y.getPersona().inserisciPerManagerResidenza(k);
        }
    }
    public static void elimina(TipoLinkResidenza y) {
        if (y != null && y.getPersona().getLinkResidenza().equals(y)) {
            ManagerResidenza k = new ManagerResidenza(y);
            y.getCitta().eliminaPerManagerResidenza(k);
            y.getPersona().eliminaPerManagerResidenza(k);
        }
    }
}
```

Come per il caso di entrambe le molteplicità 0..1, anche in questo caso la funzione `elimina()` deve prendere opportuni provvedimenti al fine di evitare di eliminare link inesistenti.

Esercizio 14: cliente

Realizzare in Java il cliente *Gestione Anagrafe*:

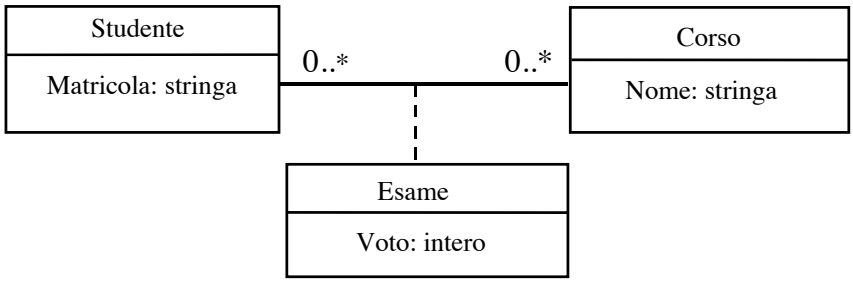
InizioSpecificaOperazioni Gestione Anagrafe

TrovaNuovi *(c: Città, a: 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

Entrambe le molteplicità sono 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 `HashSet<TipoLinkEsame>`, per poter rappresentare tutti i link;

  l'oggetto di classe `HashSet<TipoLinkEsame>` viene creato tramite il costruttore;

- hanno tre campi funzione (`inserisciLinkResidenza()`, `eliminaLinkResidenza()` e `getLinkResidenza()`) per la gestione dell'insieme dei link;

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

- hanno i due campi funzioni speciali di ausilio per `ManagerResidenza`.

Per completezza, riportiamo il codice di tutte le classi, con l'assunzione che dalla specifica si evinca che è possibile eliminare un esame una volta sostenuto.

Classe Java Studente

```
// File RespEntrambiOSTAR2/Studente.java
import java.util.*;
public class Studente {
    private final String matricola;
    private HashSet<TipoLinkEsame> insieme_link;
    public Studente(String n) {
        matricola = n;
        insieme_link = new HashSet<TipoLinkEsame>();
    }
    public String getMatricola() { return matricola; }
    public void inserisciLinkEsame(TipoLinkEsame t) {
        if (t != null && t.getStudente()==this)
            ManagerEsame.inserisci(t);
    }
    public void eliminaLinkEsame(TipoLinkEsame t) {
         if (t != null && t.getStudente()==this)
            ManagerEsame.elimina(t);
    }
     public Set<TipoLinkEsame> getLinkEsame() {
        return (HashSet<TipoLinkEsame>)insieme_link.clone();
    }
    public void inserisciPerManagerEsame(ManagerEsame a) {
        if (a != null) insieme_link.add(a.getLink());
    }
```



```
    public void eliminaPerManagerEsame(ManagerEsame a) {
        if (a != null) insieme_link.remove(a.getLink());
    }
}
```

Classe Java Corso

```
// File RespEntrambiOSTAR2/Corso.java
import java.util.*;
public class Corso {
    private final String nome;
    private HashSet<TipoLinkEsame> insieme_link;
    public Corso(String n) {
        nome = n;
        insieme_link = new HashSet<TipoLinkEsame>();
    }
    public String getNome() { return nome; }
    public void inserisciLinkEsame(TipoLinkEsame t) {
         if (t != null && t.getCorso()==this)
            ManagerEsame.inserisci(t);
    }
    public void eliminaLinkEsame(TipoLinkEsame t) {
        if (t != null && t.getCorso()==this)
            ManagerEsame.elimina(t);
    }
    public Set<TipoLinkEsame> getLinkEsame() {
        return (HashSet<TipoLinkEsame>)insieme_link.clone();
    }
    public void inserisciPerManagerEsame(ManagerEsame a) {
        if (a != null) insieme_link.add(a.getLink());
    }
```



```
    public void eliminaPerManagerEsame(ManagerEsame a) {
        if (a != null) insieme_link.remove(a.getLink());
    }
}
```

Classe Java TipoLinkEsame

```
// File 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)
        throws EccezionePrecondizioni {
        if (x == null || y == null)  // CONTROLLO PRECONDIZIONI
            throw new EccezionePrecondizioni
                ("Gli oggetti devono essere inizializzati");
        ilCorso = x; loStudente = y; voto = a;
    }
    public boolean equals(Object o) {
        if (o != null && getClass().equals(o.getClass())) {
            TipoLinkEsame b = (TipoLinkEsame)o;
            return b.ilCorso == ilCorso && b.loStudente == loStudente;
        }
        else return false;
    }
    public int hashCode() {
        return ilCorso.hashCode() + loStudente.hashCode();
    }
    public Corso getCorso() { return ilCorso; }
```



```
    public Studente getStudente() { return loStudente; }
    public int getVoto() { return voto; }
}
```

Classe Java ManagerEsame

```
// File RespEntrambi0STAR2/ManagerEsame.java

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

Si noti che, a differenza dei casi in cui almeno una delle molteplicità è 0..1, in questo caso non corriamo il rischio di creare situazioni inconsistenti con l'eliminazione di link.

Esercizio 15: cliente

Realizzare in Java il cliente *Valutazione Didattica*:

InizioSpecificaOperazioni 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

Altre molteplicità di associazione

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

1. molteplicità minima diversa da zero;

2. molteplicità massima finita.

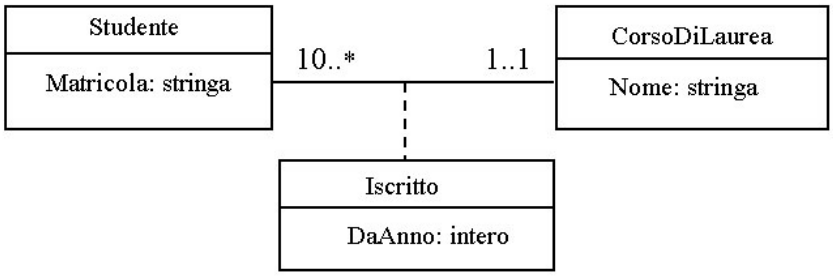
Come già chiarito nella fase di progetto, in generale prevediamo che la classe Java rispetto a cui esiste uno dei vincoli di cui sopra **abbia necessariamente responsabilità sull'associazione**. Il motivo, che verrà chiarito in seguito, è che gli oggetti di tale classe classe **devono poter essere interrogati** sul numero di link esistenti.

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*. 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 finita diversa da 1 e vincoli di molteplicità minima diversa da 0, 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.

Esempio con molteplicità arbitrarie



Prenderemo in considerazione questo esempio.

Considerazioni sulla molteplicità

Questo esempio dimostra bene 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.

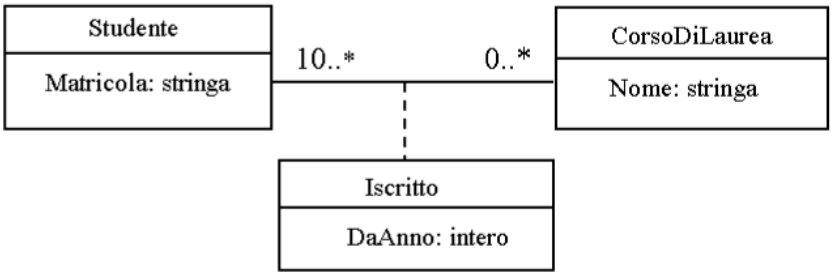
Infatti, uno studente potrebbe nascere solamente nel momento in cui esiste già un corso di laurea, ma un corso di laurea deve avere almeno dieci studenti, e questo indica una intrinseca complessità nel creare oggetti, e al tempo stesso fare in modo che essi non violino vincoli di cardinalità minima. 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

Per esigenze didattiche, affrontiamo i due casi separatamente.

Consideriamo quindi la seguente **versione semplificata** del diagramma delle classi (si notino i diversi vincoli di molteplicità).



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

Molteplicità minima diversa da zero (cont.)

- Rispetto al caso di associazione con responsabilità doppia e in cui i vincoli di molteplicità siano entrambi 0..*, la classe Java `CorsoDiLaurea` si differenzia nei seguenti aspetti:
  1. Ha un'ulteriore funzione pubblica `int quantiIscritti()`, che restituisce il numero di studenti iscritti per il corso di laurea oggetto di invocazione.

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

Molteplicità minima diversa da zero (cont.)

Rimane invece inalterata, rispetto al caso di associazione con responsabilità doppia e vincoli di molteplicità entrambi 0..*, la metodologia di realizzazione delle seguenti classi Java:

- `Studente`,
- `TipoLinkIscritto`,
- `EccezionePrecondizioni`,
- `ManagerIscritto`.

Riportiamo il codice delle classi `EccezioneCardMin` e `CorsoDiLaurea`.

Classe Java EccezioneCardMin

```
// File MoltMin/EccezioneCardMin.java

public class EccezioneCardMin extends Exception {
  private String messaggio;
  public EccezioneCardMin(String m) {
    messaggio = m;
  }
  public String toString() {
    return messaggio;
  }
}
```

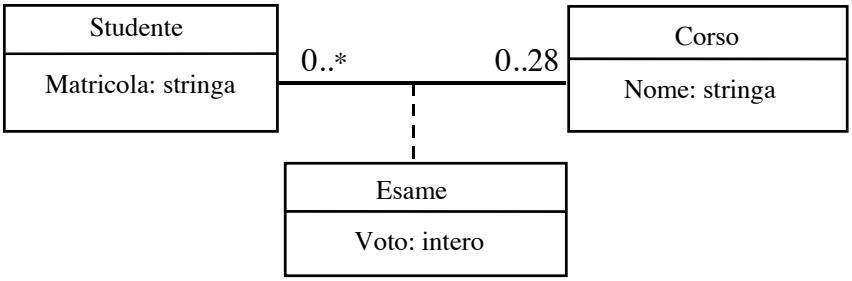
Classe Java CorsoDiLaurea

```
// File MoltMin/CorsoDiLaurea.java
import java.util.*;
public class CorsoDiLaurea {
    private final String nome;
    private HashSet<TipoLinkIscritto> insieme_link;
    public static final int MIN_LINK_ISCRITTO = 10;
           // PER IL VINCOLO DI MOLTEPLICITÀ
    public CorsoDiLaurea(String n) {
        nome = n;
        insieme_link = new HashSet<TipoLinkIscritto>();
    }
    public String getNome() { return nome; }
    public int quantiIscritti() { // FUNZIONE NUOVA
        return insieme_link.size();
    }
    public void inserisciLinkIscritto(TipoLinkIscritto t) {
      if (t != null && t.getCorsoDiLaurea()==this)
         ManagerIscritto.inserisci(t);
    }
    public void eliminaLinkIscritto(TipoLinkIscritto t) {
      if (t != null && t.getCorsoDiLaurea()==this)
         ManagerIscritto.elimina(t);
    }
    public Set<TipoLinkIscritto> getLinkIscritto() throws EccezioneCardMin {
```



```
        if (quantiIscritti() < MIN_LINK_ISCRITTO)
            throw new EccezioneCardMin("Cardinalita' minima violata");
        else return (HashSet<TipoLinkIscritto>)insieme_link.clone();
    }
    public void inserisciPerManagerIscritto(ManagerIscritto a) {
        if (a != null) insieme_link.add(a.getLink());
    }
    public void eliminaPerManagerIscritto(ManagerIscritto a) {
        if (a != null) insieme_link.remove(a.getLink());
    }
}
```

Molteplicità massima finita



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

Molteplicità massima finita (cont.)

- Rispetto al caso di associazione con responsabilità doppia e in cui i vincoli di molteplicità siano entrambi 0..*, la classe Java `Studente` si differenzia nei seguenti aspetti:
  1. Ha un'ulteriore funzione pubblica `int quantiEsami()`, che restituisce il numero di esami sostenuti dallo studente oggetto di invocazione.

     In questa maniera, il cliente si può rendere conto se sia possibile inserire un nuovo esame senza violare i vincoli di molteplicità oppure no.
  2. La funzione `int getLinkEsami()` lancia una opportuna eccezione (di tipo `EccezioneCardMax`) quando l'oggetto di invocazione non rispetta il vincolo di cardinalità massima sull'associazione *Iscritto*.

Molteplicità massima finita (cont.)

Rimangono invece inalterate, rispetto al caso di associazione con responsabilità doppia e vincoli di molteplicità entrambi 0..* le seguenti classi Java:

- `Corso`,
- `TipoLinkEsame`,
- `EccezionePrecondizioni`,
- `ManagerEsame`.

Riportiamo il codice delle classi `EccezioneCardMax` e `Studente`.

Classe Java EccezioneCardMax

```
// File MoltMax/EccezioneCardMax.java

public class EccezioneCardMax extends Exception {
  private String messaggio;
  public EccezioneCardMax(String m) {
    messaggio = m;
  }
  public String toString() {
    return messaggio;
  }
}
```

Classe Java Studente

```
// File MoltMax/Studente.java

import java.util.*;

public class Studente {
    private final String matricola;
    private HashSet<TipoLinkEsame> insieme_link;
    public static final int MAX_LINK_ESAME = 28;
    // PER IL VINCOLO DI MOLTEPLICITÀ
    public Studente(String n) {
        matricola = n;
        insieme_link = new HashSet<TipoLinkEsame>();
    }
    public String getMatricola() { return matricola; }
    public int quantiEsami() { // FUNZIONE NUOVA
        return insieme_link.size();
    }
    public void inserisciLinkEsame(AssociazioneEsame a)  {
        if (a != null) insieme_link.add(a.getLink());
    }
    public void eliminaLinkEsame(AssociazioneEsame a) {
        if (a != null) insieme_link.remove(a.getLink());
    }
    public Set<TipoLinkEsame> getLinkEsame() throws EccezioneCardMax {
```



```
        if (insieme_link.size() > MAX_LINK_ESAME)
            throw new EccezioneCardMax("Cardinalita' massima violata");
        else return (HashSet<TipoLinkEsame>)insieme_link.clone();
    }
}
```

Esercizio 16

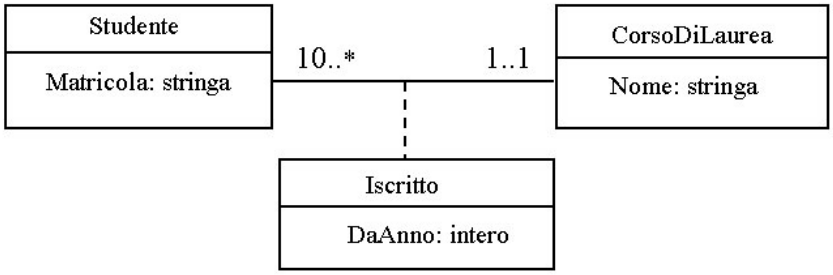
Con riferimento all'esercizio 15, realizzare in Java il cliente *Valutazione Didattica* per il caso corrente.

Molteplicità massima 1

Un caso particolare di molteplicità massima finita si ha quando essa sia pari a 1. In tal caso dobbiamo gestire l'associazione secondo il modello e le strutture di dati visti in precedenza per il vincolo di molteplicità 0..1.

In particolare, dobbiamo prevedere gli opportuni controlli per la classe che gestisce l'associazione.

Molteplicità massima 1 (cont.)



Prenderemo in considerazione il diagramma delle classi già visto in precedenza, nella sua versione **non semplificata**.

Notiamo che dai vincoli di molteplicità si evince che sia *Studente* sia *Corso* hanno responsabilità sull'associazione.

Classe Java Studente

```
// File MoltMax1/Studente.java
public class Studente {
    private final String matricola;
    private TipoLinkIscritto link;
    public static final int MIN_LINK_ISCRITTO = 1;
    public Studente(String n) { matricola = n; }
    public String getMatricola() { return matricola; }
    public int quantiIscritti() { // FUNZIONE NUOVA
        if (link == null)
            return 0;
        else return 1;
    }
    public void inserisciLinkIscritto(TipoLinkIscritto t) {
      if (t != null && t.getStudente()==this)
         ManagerIscritto.inserisci(t);
    }
    public void eliminaLinkIscritto(TipoLinkIscritto t) {
      if (t != null && t.getStudente()==this)
         ManagerIscritto.elimina(t);
    }
    public TipoLinkIscritto getLinkIscritto() throws EccezioneCardMin {
        if (link == null)
            throw new EccezioneCardMin("Cardinalita' minima violata");
        else
```

```
            return link;
    }
    public void inserisciPerManagerIscritto(ManagerIscritto a) {
        if (a != null) link = a.getLink();
    }
    public void eliminaPerManagerIscritto(ManagerIscritto a) {
        if (a != null) link = null;
    }
}
```

Classe Java ManagerIscritto

```
// File MoltMax1/ManagerIscritto.java
public final class ManagerIscritto {
    private ManagerIscritto(TipoLinkIscritto x) { link = x; }
    private TipoLinkIscritto link;
    public TipoLinkIscritto getLink() { return link; }
    public static void inserisci(TipoLinkIscritto y) {
        if (y != null && y.getStudente().quantiIscritti() == 0) {
            ManagerIscritto k = new ManagerIscritto(y);
            k.link.getCorsoDiLaurea().inserisciPerManagerIscritto(k);
            k.link.getStudente().inserisciPerManagerIscritto(k);
        }
    }
    public static void elimina(TipoLinkIscritto y) {
        try {
            if (y != null && y.getStudente().getLinkIscritto().equals(y) ) {
                ManagerIscritto k = new ManagerIscritto(y);
                k.link.getCorsoDiLaurea().eliminaPerManagerIscritto(k);
                k.link.getStudente().eliminaPerManagerIscritto(k);
            }
        }
        catch (EccezioneCardMin e) {
            System.out.println(e);
        }
    }
}
```

Molteplicità massima 1 (cont.)

Notiamo che la classe `Studente` ha ovviamente un campo dato di tipo `TipoLinkIscritto` (e non `Set`) e che la classe `ManagerIscritto` deve effettuare i controlli del caso, in particolare:

- che l'inserimento avvenga solo se lo studente non è iscritto (sfruttando la funzione `quantiIscritti()` di `Studente`),
- che la cancellazione avvenga solo per link esistenti.

Rimangono invece inalterate le seguenti classi Java:

- `TipoLinkIscritto/EccezionePrecondizioni` (realizzate come sempre),
- `CorsoDiLaurea` (realizzata come nel caso –semplificato– visto in precedenza in cui lo studente può essere iscritto ad un numero qualsiasi di corsi di laurea).

Esercizio 17: cliente

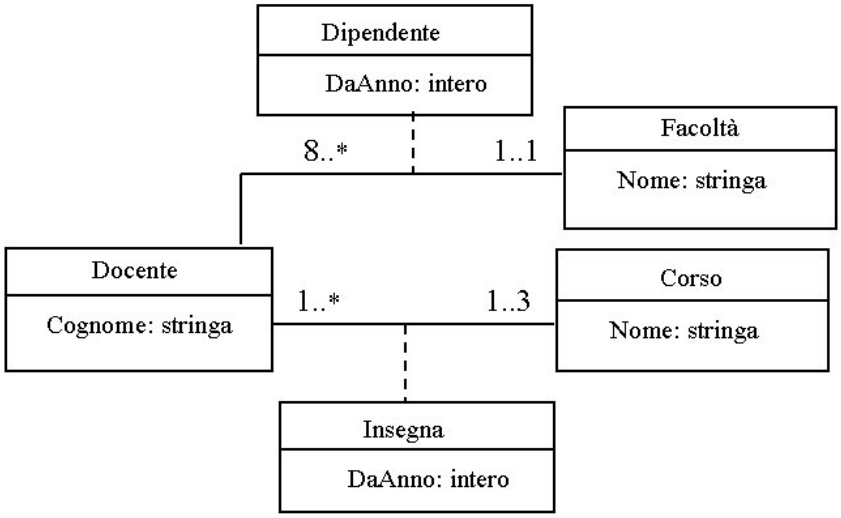
Realizzare in Java il cliente *Cambiamento Indirizzo*:

InizioSpecificaOperazioni Cambiamento Indirizzo

Cambiano *(i: Insieme(Studente), c: CorsoDiLaurea, a: intero)*
pre: nessuna
post: ogni studente di *i* si iscrive al corso di laurea *c* a partire dall'anno *a*

FineSpecifica

Esercizio 18



Realizzare in Java questo diagramma delle classi.

Associazioni n-arie

Si trattano generalizzando quanto visto per le associazioni binarie.

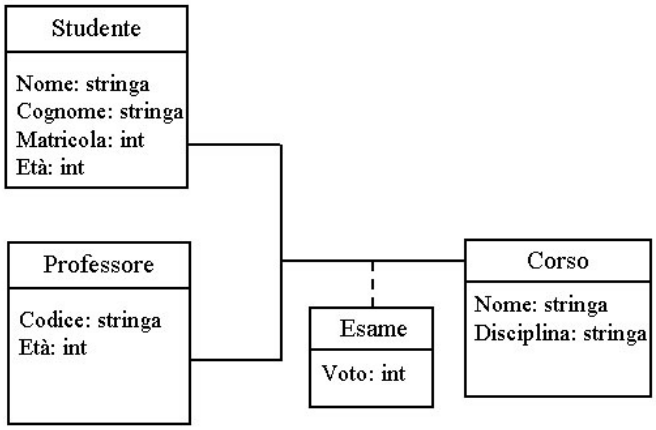
Ricordiamo che noi assumiamo che le molteplicità delle associazioni n-arie siano sempre 0..*.

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

Nel caso di responsabilità di una sola classe, si prevede la struttura di dati per rappresentare i link solo in quella classe.

Nel caso di responsabilità di più classi, si definisce anche la classe `ManagerA`, secondo le regole viste per le associazioni binarie.

Esempio



A titolo di esempio, ci occuperemo della realizzazione in Java di questo diagramma delle classi.

Esempio (cont.)

- Assumiamo che la fase di progetto abbia stabilito che la responsabilità sull'associazione *Esame* sia delle classi UML *Studente* e *Professore*.
- Classi Java:
  - `TipoLinkEsame/EccezionePrecondizione`;
  - `ManagerOccupazione`;
  - `Studente/Professore`;
  - `Corso`.

Esercizio 19

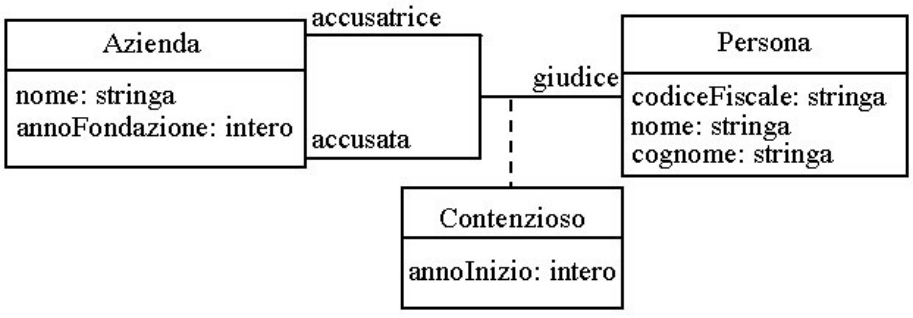
1. Realizzare tutte le classi Java per l'esempio corrente.

2. Con riferimento all'esercizio 15, realizzare in Java l'operazione *StudenteBravo* del cliente *Valutazione Didattica* per il caso corrente.

3. Arricchire il cliente con la seguente operazione, e realizzarla in Java:

   ProfessoreBuono *(p: Professore): booleano*
   pre: nessuna
   post: *result* è *true* se e solo se tutti gli esami sostenuti con il professore *p* sono stati superati con voto non inferiore a 27

Esercizio 20



Realizzare in Java questo diagramma delle classi, assumendo che la fase di progetto abbia stabilito che la responsabilità sull'associazione *Contenzioso* sia delle classi UML *Azienda* (in **entrambi** i ruoli) e *Persona*.

Esercizio 21

Con riferimento all'esercizio precedente realizzare le seguenti operazioni.

L'ufficio giudiziario del lavoro deve poter effettuare, come cliente della nostra applicazione, dei controlli sui contenziosi. A questo scopo, si faccia riferimento alle seguenti operazioni:

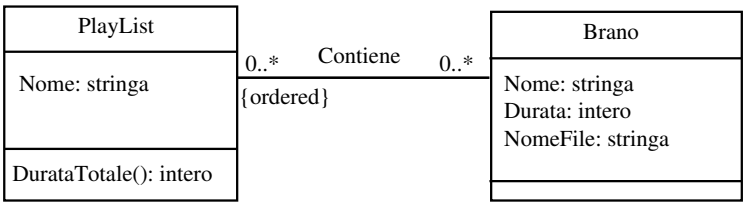
- Dato un insieme S di aziende, restituire il sottoinsieme di S formato da tutte le aziende che sono accusatrici in almeno due contenziosi con la stessa azienda accusata.
- Data un'azienda A ed una persona P, dire se A è accusata in almeno un contenzioso in cui P opera come giudice.

Associazioni Ordinate

- Le associazioni ordinate si realizzano in modo del tutto analogo alle relazioni non ordinate.
- Per mantenere l'ordinamento però si fa uso delle classi `List` e `LinkedList` invece di `Set` e `HashSet`.
- Si noti che l'ordinamento sulle associazini è realizzato con `List` e `LinkedList` e non con una struttura dati per insiemi ordinati come `OrderedSet` e `TreeSet`, perchè l'ordinamento non nasce da proprieta degli elementi dell'insieme (cioè dei link) ma viene definito esternamente ed essi come appunto succede quando mettiamo degli elementi in una lista.
- Si noti inoltre che nel memorizzare i link in una lista dobbiamo stare attenti a non avere ripetizioni perchè in una associazione, anche se ordinata, non ci possono essere due link uguali.

Esempio con responsabilità singola

Consideriamo il seguente diagramma delle classi.



Assumiamo di avere stabilito, nella fase di progetto, che:

- il nome di una playlist, e il nome, la durata ed il nome del file associati ad un un brano **non cambiano**;
- **solo PlayList ha responsabilità** sull'associazione (ci interessa conoscere quali brani sono contenuti in una playlist, ma non non ci interessa conoscere le playlist che contengono un dato brano).

Specifica della classe UML PlayList

InizioSpecificaClasse PlayList

durataTotale (): *intero*
pre: nessuna
post: *result* è pari alla somma delle durate dei brani contenuti in *this*

FineSpecifica

Realizzazione in Java della classe PlayList

```
// File OrdinateOSTAR/PlayList.java
import java.util.*;
public class PlayList {
    private final String nome;
    private LinkedList<Brano> insieme_link;
    public PlayList(String n) {
        nome = n;
        insieme_link = new LinkedList<Brano>();
    }
    public String getNome() { return nome; }
    public void inserisciLinkContiene(Brano b) {
        if (b != null && !insieme_link.contains(b)) insieme_link.add(b);
    }
    public void eliminaLinkContiene(Brano b) {
        if (b != null) insieme_link.remove(b);
    }
    public List<Brano> getLinkContiene() {
        return (LinkedList<Brano>)insieme_link.clone();
    }

    public int durataTotale() {
        int result = 0;
        Iterator<Brano> ib = insieme_link.iterator();
        while (ib.hasNext()) {
```



```
            Brano b = ib.next();
            result = result + b.getDurata();
        }
        return result;
    }
}
```

Realizzazione in Java della classe Brano

```
// File OrdinateOSTAR/Brano.java

public class Brano {
  private final String nome;
  private final int durata;
  private final String nomefile;
  public Brano(String n, int d, String f) {
    nome = n;
    durata = d;
    nomefile = f;
  }
  public String getNome() { return nome; }
  public int getDurata() { return durata; }
  public String getNomeFile() { return nomefile; }
}
```


Un cliente

Realizziamo ora in Java il cliente *Analisi PlayList*, specificato di seguito:

InizioSpecificaCliente Analisi PlayList

PiùLunghe (*i: Insieme(PlayList)*): *Insieme(PlayList)*
pre: nessuna
post: *result* è costituito dalle PlayList di *i* la cui durata totale è massima

FineSpecifica

Realizzazione in Java del cliente

La progettazione dell'algoritmo è lasciata come esercizio

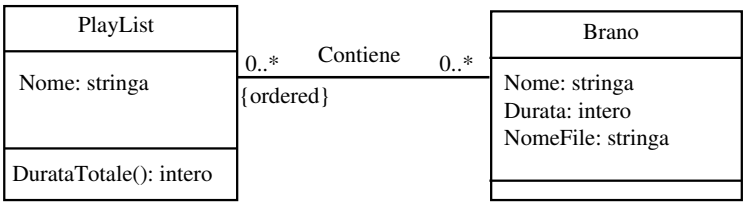
```
// File OrdinateOSTAR/AnalisiPlayList.java
import java.util.*;

public final class AnalisiPlayList {
  public static Set<PlayList> piuLunghe(Set<PlayList> i) {
    HashSet<PlayList> result = new HashSet<PlayList>();
    int duratamax = maxDurata(i);
    Iterator<PlayList> it = i.iterator();
    while(it.hasNext()) {
      PlayList pl = it.next();
      int durata = pl.durataTotale();
      if (durata == duratamax)
        result.add(pl);
    }
    return result;
  }
  private static int maxDurata(Set<PlayList> i) {
    int duratamax = 0;
    Iterator<PlayList> it = i.iterator();
    while(it.hasNext()) {
      PlayList pl = it.next();
      int durata = pl.durataTotale();
```



```
      if (durata > duratamax)
        duratamax = durata;
    }
    return duratamax;
  }
}
```

Esempio responsabilità doppia

Consideriamo lo stesso diagramma delle classi visto in precedenza.



Assumiamo di avere stabilito, nella fase di progetto, che:

- il nome di una playlist, e il nome, la durata ed il nome del file associati ad un brano **non cambiano**;
- sia **PlayList che Brano hanno responsabilità** sull'associazione (ci interessa conoscere sia quali brani sono contenuti in una playlist, che le playlist che contengono un dato brano).

Esempio responsabilità doppia (cont.)

- In questo caso dobbiamo adattare la metodologia generale, prevedendo:
  - la realizzazione della classe `TipoLinkContiene`,
  - la realizzazione della classe `ManagerContiene`,
  - che la classe `Brano` abbia un campo dato di tipo `HashSet`, per rappresentare la struttura di dati non ordinata,
  - che la classe `PlayList` abbia un campo dato di tipo `LinkedList`, per rappresentare la struttura di dati ordinata.

Realizzazione in Java della classe PlayList

```
// File OrdinateEntrambiOSTAR/PlayList.java
import java.util.*;
public class PlayList {
    private final String nome;
    private LinkedList<TipoLinkContiene> insieme_link;
    public PlayList(String n) {
        nome = n;
        insieme_link = new LinkedList<TipoLinkContiene>();
    }
    public String getNome() { return nome; }
    public void inserisciLinkContiene(TipoLinkIscritto t) {
      if (t != null && t.getPlayList()==this)
         ManagerContiene.inserisci(t);
    }
    public void eliminaLinkContiene(TipoLinkIscritto t) {
      if (t != null && t.getPlayList()==this)
         ManagerContiene.elimina(t);
    }
    public List<TipoLinkContiene> getLinkContiene() {
        return (LinkedList<TipoLinkContiene>)insieme_link.clone();
    }
    public void inserisciPerManagerContiene(ManagerContiene a){
        if (a != null && !insieme_link.contains(a.getLink()))
            insieme_link.add(a.getLink());
```



```
    }
    public void eliminaPerManagerContiene(ManagerContiene a) {
        if (a != null) insieme_link.remove(a.getLink());
    }
    public int durataTotale() {
        int result = 0;
        Iterator<TipoLinkContiene> il = insieme_link.iterator();
        while (il.hasNext()) {
            Brano b = il.next().getBrano();
            result = result + b.getDurata();
        }
        return result;
    }
}
```

Realizzazione in Java della classe Brano

```
// File OrdinateEntrambiOSTAR/Brano.java
import java.util.*;
public class Brano {
    private final String nome;
    private final int durata;
    private final String nomefile;
    private HashSet<TipoLinkContiene> insieme_link;
    public Brano(String n, int d, String f) {
        nome = n;
        durata = d;
        nomefile = f;
        insieme_link = new HashSet<TipoLinkContiene>();
    }
    public String getNome() { return nome; }
    public int getDurata() { return durata; }
    public String getNomeFile() { return nomefile; }
    public void inserisciLinkContiene(TipoLinkContiene t) {
      if (t != null && t.getBrano()==this)
          ManagerContiene.inserisci(t);
    }
    public void eliminaLinkContiene(TipoLinkContiene t) {
      if (t != null && t.getBrano()==this)
          ManagerContiene.elimina(t);
    }
```



```
    public Set<TipoLinkContiene> getLinkContiene() {
        return (HashSet<TipoLinkContiene>)insieme_link.clone();
    }
    public void inserisciPerManagerContiene(ManagerContiene a){
        if (a != null) insieme_link.add(a.getLink());
    }
    public void eliminaPerManagerContiene(ManagerContiene a) {
        if (a != null) insieme_link.remove(a.getLink());
    }
}
```

Realizzazione della classe TipoLinkContiene

```
// File OrdinateEntrambiOSTAR/TipoLinkContiene.java
public class TipoLinkContiene {
    private final PlayList laPlayList;
    private final Brano ilBrano;
    public TipoLinkContiene(PlayList x, Brano y)
        throws EccezionePrecondizioni {
        if (x == null || y == null)  // CONTROLLO PRECONDIZIONI
            throw new EccezionePrecondizioni
                ("Gli oggetti devono essere inizializzati");
        laPlayList = x; ilBrano = y;
    }
    public boolean equals(Object o) {
        if (o != null && getClass().equals(o.getClass())) {
            TipoLinkContiene b = (TipoLinkContiene)o;
            return b.ilBrano == ilBrano && b.laPlayList == laPlayList;
        }
        else return false;
    }
    public int hashCode() {
        return laPlayList.hashCode() + ilBrano.hashCode();
    }
    public PlayList getPlayList() { return laPlayList; }
    public Brano getBrano() { return ilBrano; }
}
```

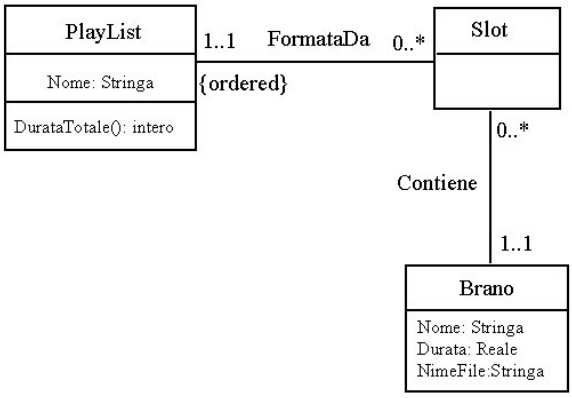

Realizzazione della classe ManagerContiene

```
// File OrdinateEntrambiOSTAR/ManagerContiene.java

public final class ManagerContiene {
    private ManagerContiene(TipoLinkContiene x) { link = x; }
    private TipoLinkContiene link;
    public TipoLinkContiene getLink() { return link; }
    public static void inserisci(TipoLinkContiene y) {
        if (y != null) {
            ManagerContiene k = new ManagerContiene(y);
            y.getPlayList().inserisciPerManagerContiene(k);
            y.getBrano().inserisciPerManagerContiene(k);
        }
    }
    public static void elimina(TipoLinkContiene y) {
        if (y != null) {
            ManagerContiene k = new ManagerContiene(y);
            y.getPlayList().eliminaPerManagerContiene(k);
            y.getBrano().eliminaPerManagerContiene(k);
        }
    }
}
```

Esercizio 22

Realizzare il seguente diagramma delle classi (che ci consente la ripetizione dei brani), tenendo cont

- **solo Slc** sati a sapere quali slo
- sia *Playl*



Esercizio 22 (cont.)

Nella realizzazione, possiamo tenere conto che (cfr. *Seconda Parte*) la classe Slot non è presente esplicitamente nei requisiti, ed è stata introdotta per poter distinguere la posizione dei brani dai brani stessi.

Di conseguenza in realtà **non siamo interessati a rappresentare esplicitamente Slot**, in quanto nella nostra applicazione non abbiamo mai bisogno di riferirci ad oggetti *Slot*.

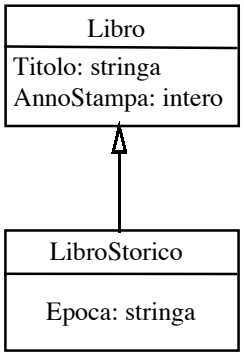
Realizzazione di generalizzazioni

Nell'esposizione di questo argomento, seguiremo quest'ordine:

- relazione is-a fra due classi;
- specializzazione di operazioni;
- generalizzazioni disgiunte e complete.

Generalizzazione

Affrontiamo ora il caso in cui abbiamo una generalizzazione nel diagramma delle classi. Ci riferiamo 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 ai clienti 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.

Ripasso: livelli di accesso nelle classi Java

Un campo di una classe (dato, funzione o classe) può essere specificato con uno fra **quattro** livelli di accesso:

A. `public`,

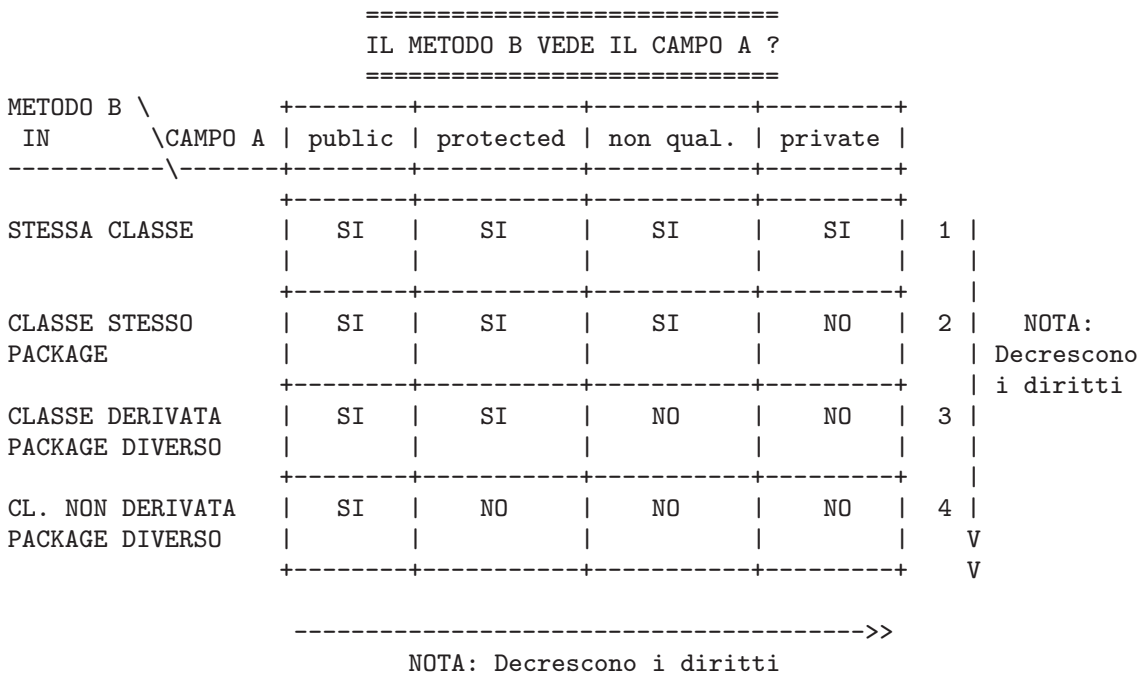
B. `protected`,

C. non qualificato (è il *default*, intermedio fra protetto e privato),

D. `private`.

Anche un'intera classe `C` (solo se **non è interna ad altra classe**) può essere dichiarata `public`, ed in tale caso la classe deve essere dichiarata nel file `C.java`.

Classi: regole di visibilità



Information hiding e generalizzazione (cont.)

Occorre 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 Java che ha campi protetti (tipicamente, ciò avviene quando la corrispondente classe UML fa parte di una gerarchia).

Generalizzazione e strutturazione in package

In particolare, seguiremo le seguenti regole:

- continueremo per il momento ad assumere di lavorare con il package senza nome (più avanti torneremo su questo aspetto);
- per ogni classe Java *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 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;
  }
}
```


Costruttori di classi derivate: ripasso

Comportamento di un costruttore di una classe `D` derivata da `B`:

1. **se** ha come prima istruzione `super()`, allora viene chiamato il costruttore di `B` esplicitamente invocato;

   altrimenti viene chiamato il costruttore senza argomenti di `B`;

2. viene eseguito il corpo del costruttore.

Questo vale **anche per il costruttore standard** di `D` senza argomenti (come al solito, disponibile se e solo se in `D` non vengono definiti esplicitamente costruttori).

La classe Java LibroStorico

```
// File 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

InizioSpecificaCliente Valutazione Biblioteca

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

FineSpecifica

Gli algoritmi vengono lasciati per esercizio.

Realizzazione del cliente

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

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



```
        return quanti;
    }
    private ValutazioneBiblioteca() { }
}
```

Riassunto struttura file e direttori

```
|
+---Generalizzazione
|   |   MainBiblio.java
|   |   ValutazioneBiblioteca.java
|   |
|   +---Libro
|   |       Libro.java
|   |
|   \---LibroStorico
|           LibroStorico.java
```

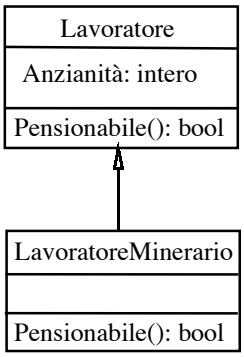

Ridefinizione

Nella classe derivata è possibile fare **overriding** (dall'inglese, *ridefinizione*, *sovrascrittura*) delle funzioni della classe base.

Fare overriding di una funzione `f()` della classe base `B` vuol dire definire nella classe derivata `D` una funzione `f()` **in cui sono uguali il numero e il tipo degli argomenti**, mentre **il tipo di ritorno deve essere identico**.

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: classe base

```
// File 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: classe derivata

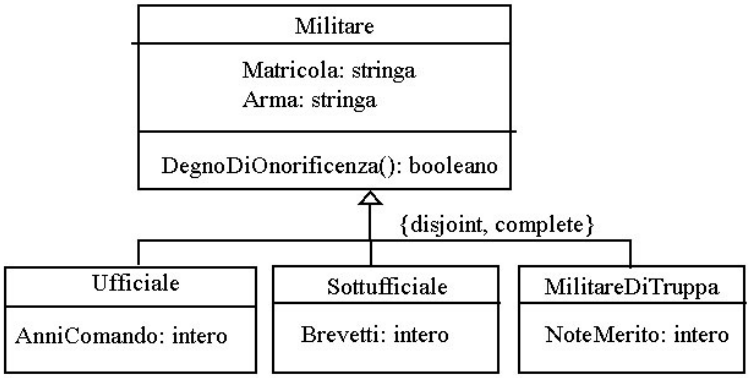
```
// File Generalizzazione/LavoratoreMinerario/LavoratoreMinerario.java

package LavoratoreMinerario;
import Lavoratore.*;

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


Generalizzazioni disgiunte e complete

Poiché 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 mostreremo in un esempio.

Generalizzazioni disgiunte e complete (cont.)



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", 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 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 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 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 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;
  }
}
```

Organizzazione in packages

Per evitare ogni potenziale conflitto sull'uso degli identificatori di classe, è possibile strutturare i file sorgente in package.

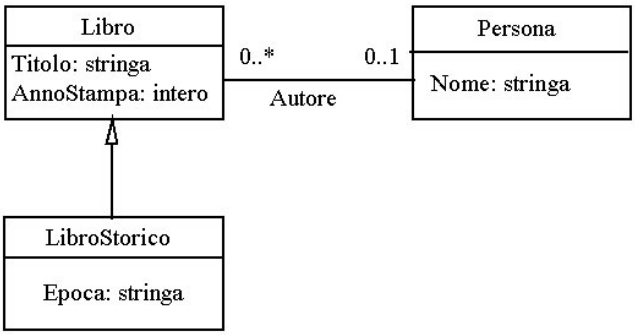
Una regola possibile è la seguente:

- Tutta l'applicazione viene messa in un package Java `P`, nel direttorio `P`.
- Ogni classe Java dell'applicazione proveniente dal diagramma delle classi (anche quelle definite per le associazioni) viene messa nel package `P`.
- Ciò vale anche per quelle definite per i tipi, a meno che siano in opportuni direttori resi accessibili mediante la variabile d'ambiente `classpath`.

Organizzazione in packages (cont.)

- Nel caso di classi con campi protetti, vanno previsti sottodirettori e sottopackage, come visto in precedenza.

Packages, esempio



Supponiamo che solamente *Libro* abbia responsabilità sull'associazione *Autore*.

Packages, esempio (cont.)

InizioSpecificaCliente StatisticaAutori

Prolifici (*i: Insieme(Libro)*): *Insieme(Persona)*
pre: nessuna
post: *result* è l'insieme di persone che sono autori di almeno due libri fra quelli di *i*

FineSpecifica

Struttura file e direttori

```
|
+---PackageLibri
|   |   StatisticaAutori.java
|   |   Persona.java
|   |   Test
|   |   MainLibri.java
|   |
|   +---Libro
|   |       Libro.java
|   |
|   \---LibroStorico
|           LibroStorico.java
|
```


La classe Java Persona

```
// File PackageLibri/Persona.java

package PackageLibri;

public class Persona {
    private final String nome;
    public Persona(String n) {
        nome = n;
    }
    public String getNome() {
        return nome;
    }
    public String toString() {
        return nome ;
    }
}
```

La classe Java Libro

```
// File PackageLibri/Libro/Libro.java

package PackageLibri.Libro;
import PackageLibri.*;

public class Libro {
  protected final String titolo;
  protected final int annoStampa;
  protected Persona autore;
  public Libro(String t, int a) { titolo = t; annoStampa = a;}
  public void setAutore(Persona p) { autore = p; }
  public Persona getAutore() { return autore; }
  public String getTitolo() { return titolo; }
  public int getAnnoStampa() { return annoStampa; }
  public String toString() {
    return titolo +
      (autore != null ? ", di " + autore.toString() : ", Anonimo") +
      ", dato alle stampe nel " + annoStampa;
  }
}
```

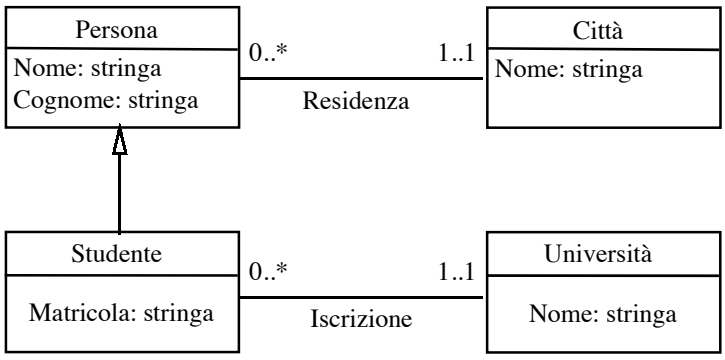

La classe Java LibroStorico

```
// File PackageLibri/LibroStorico/LibroStorico.java

package PackageLibri.LibroStorico;
import PackageLibri.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;
  }
}
```

La classe Java StatisticaAutori

```
// File PackageLibri/StatisticaAutori.java

package PackageLibri;
import PackageLibri.Libro.*;
import PackageLibri.LibroStorico.*;
import java.util.*;
public final class StatisticaAutori {
    public static Set<Persona> prolifici(Set<Libro> ins) {
        Set<Persona> result = new HashSet<Persona>();
        System.out.println
            ("La funzione prolifici() e' da implementare per esercizio!");
        return result;
    }
    private StatisticaAutori() { }
}
```


Esercizio 23



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 24

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.

Aspetti di UML non trattati in dettaglio

Per mancanza di tempo, nella fase di realizzazione non possiamo trattare in dettaglio alcuni aspetti di UML visti nella fase di analisi:

- generalizzazioni non disgiunte;
- specializzazione di attributi;
- specializzazione di associazioni;
- ereditarietà multipla.

Tuttavia si vedano esercizi e soluzioni sul sito del corso.

Soluzioni degli esercizi della quarta parte

Soluzione esercizio 1

Per l'operazione **Convivono** adottiamo il seguente algoritmo:

```
Insieme(stringa) telefoni_p1 = p1.numTel;
Insieme(stringa) telefoni_p2 = p2.numTel;
per ogni stringa elem di telefoni_p1
    se elem appartiene a telefoni_p2
        allora return true;
return false;
```

L'algoritmo viene realizzato tramite la funzione `convivono()` della seguente classe Java.

```
// File MolteplicitaAttributi/AnalisiRecapiti.java
import java.util.*;
public final class AnalisiRecapiti {
  public static boolean convivono(Persona p1, Persona p2) {
    Set<String> telefoni_p1 = p1.getNumTel();
    Set<String> telefoni_p2 = p2.getNumTel();
    Iterator<String> it = telefoni_p1.iterator();
    while(it.hasNext()) {
      String elem = it.next();
      if (telefoni_p2.contains(elem))
        return true;
    }
    return false;
  }
  private AnalisiRecapiti() {};
}
```

Soluzione esercizio 2

```
// File MolteplicitaAttributiCond/GestioneRubrica.java
import java.util.*;
import IteratoreSolaLettura.*;

public final class GestioneRubrica {
    public static Set<String> tuttiNumTel (Persona p1, Persona p2) {
        HashSet<String> result = new HashSet<String>();
        Iterator<String> it1 = p1.getNumTel();
        Iterator<String> it2 = p2.getNumTel();
        while(it1.hasNext())
            result.add(it1.next());
        while(it2.hasNext())
            result.add(it2.next());
        return result;
    }
    private GestioneRubrica() { };
}

// File MolteplicitaAttributiCond/AnalisiRecapiti.java
import java.util.*;
import IteratoreSolaLettura.*;
public final class AnalisiRecapiti {
    public static boolean convivono(Persona p1, Persona p2) {
        Iterator<String> it1 = p1.getNumTel();
```



```
        while(it1.hasNext()) {
            String elem1 = it1.next();
            Iterator<String> it2 = p2.getNumTel();
            while(it2.hasNext()) {
                String elem2 = it2.next();
                if (elem1.equals(elem2))
                    return true;
            }
        }
        return false;
    }
    private AnalisiRecapiti() {};
}
```

Soluzione esercizio 3

```
// File AttributiEOperazioni/Esercizio/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();
    }
}
```

Soluzione esercizio 4

Per l'operazione **EtàMediaRicchi** adottiamo il seguente algoritmo di cui viene dato il primo raffinamento; raffinamenti successivi sono lasciati per esercizio.

```
1. trova l'aliquota massima fra le persone in i
2. seleziona le persone in i con l'aliquota massima,
   contandole e sommandone le età in mesi
3. restituisci la media delle età delle persone selezionate
```

L'algoritmo viene realizzato tramite la funzione `etaMediaRicchi()` della seguente classe Java.

```
// File AttributiEOperazioni/Esercizio/AnalisiRedditi.java
import java.util.*;
public final class AnalisiRedditi {
  public static double etaMediaRicchi(Set<Persona> i, int g, int m, int a) {
    int aliquotaMassima = 0;
    Iterator<Persona> it = i.iterator();
    while(it.hasNext()) {
      Persona elem = it.next();
      if (elem.aliquota() > aliquotaMassima)
        aliquotaMassima = elem.aliquota();
    }
    int quantiRicchi = 0;
    double sommaEtaRicchi = 0.0;
    it = i.iterator();
```

```
    while(it.hasNext()) {
      Persona elem = it.next();
      if (elem.aliquota() == aliquotaMassima) {
        sommaEtaRicchi += elem.eta(g,m,a);
        quantiRicchi++;
      }
    }
    return sommaEtaRicchi / quantiRicchi;
  }
  private AnalisiRedditi() {}
}
```

Soluzione esercizio 5

Per l'operazione **RedditoMedioInGrandiAziende** adottiamo il seguente algoritmo:

```
int quantiInGrandiAziende = 0;
double sommaRedditoDipendentiGrandiAziende = 0.0;
per ogni Persona elem di i
    se elem lavora in una grande azienda
        allora quantiInGrandiAziende++;
               sommaRedditoDipendentiGrandiAziende += elem.Reddito;
return sommaRedditoDipendentiGrandiAziende / quantiInGrandiAziende;
```

L'algoritmo viene realizzato tramite la funzione `redditoMedioInGrandiAziende()` della seguente classe Java.

```
// File Associazioni01/AnalisiAziende.java

import java.util.*;

public final class AnalisiAziende {
    public static double redditoMedioInGrandiAziende (Set<Persona> i) {
        int quantiInGrandiAziende = 0;
        double sommaRedditoDipendentiGrandiAziende = 0.0;
        Iterator<Persona> it = i.iterator();
        while(it.hasNext()) {
            Persona elem = it.next();
            if (elem.getLavoraIn() != null &&
```

```
                elem.getLavoraIn().dimensione().equals("Grande")) {
                quantiInGrandiAziende++;
                sommaRedditoDipendentiGrandiAziende += elem.getReddito();
            }
        }
        return sommaRedditoDipendentiGrandiAziende / quantiInGrandiAziende;
    }
    private AnalisiAziende() { }
}
```

Soluzione esercizio 6

Per l'operazione **ControllataDaSeStessa** adottiamo il seguente algoritmo **ricorsivo**:

```
Insieme(Azienda) controllanti = insieme vuoto;
se a non esiste
    allora return false;
se a appartiene a controllanti
    allora return true;
altrimenti
    inserisci a in controllanti
    return controllataDaSeStessa(a.controllante)
```

L'algoritmo viene realizzato tramite la funzione `controllataDaSeStessa()` della seguente classe Java. Si noti la presenza della funzione ausiliaria privata `controllataRicorsiva()`.

```
// File Ruoli/RicognizioneTruffe.java

import java.util.*;

public final class RicognizioneTruffe {
    private static HashSet<Azienda> controllanti;
    public static boolean controllataDaSeStessa (Azienda a) {
        controllanti = new HashSet<Azienda>();
        return controllataRicorsiva(a);
    }
    private static boolean controllataRicorsiva (Azienda a) {
```

```
        if (a == null)
            return false;
        else
            if (controllanti.contains(a))
                return true;
            else {
                controllanti.add(a);
                return controllataRicorsiva(a.getControllante());
            }
    }
    private RicognizioneTruffe() { }
}
```

Soluzione esercizio 7

Per l'operazione **PiuDipendenti** adottiamo il seguente algoritmo di cui viene dato il primo raffinamento; raffinamenti successivi sono lasciati per esercizio.

1. calcola il numero n di aziende distinte per cui almeno una fra le persone di i ha lavorato
2. crea un vettore vettore_aziende di n elementi in cui inserisci le n aziende distinte
3. crea un vettore conta_persone di n interi inizializzati a 0
4. memorizza in conta_persone quante persone hanno lavorato per ciascuna azienda, utilizzando l'indice di vettore_aziende
5. trova in conta_persone l'indice indice_max dell'elemento che contiene il massimo valore
6. return vettore_aziende[indice_max]

L'algoritmo viene realizzato tramite la funzione `piuDipendenti()` della seguente classe Java. Si noti la presenza di una funzione di servizio `private`.

```
// File AssOSTAR/AnalisiCollocamento.java

import java.util.*;

public final class AnalisiCollocamento {
    private static Object[] vettore_aziende;
    private static int[] conta_persone;
    private static int n;
```

```
    private static int indiceAzienda(Azienda az) {
        // funzione di servizio
        // restituisce l'indice di az nel vettore vettore_aziende
        int result = 0;
        for (int i = 0; i < n; i++)
            if (vettore_aziende[i] == az)
                result = i;
        return result;
    }

    public static Azienda piuDipendenti (Set<Persona> i) {
        /* 1 */
        HashSet<Azienda> insieme_aziende = new HashSet<Azienda>();
        Iterator<Persona> it = i.iterator();
        while(it.hasNext()) {
            Persona elem = it.next();
            Set<Azienda> az_elem = elem.getLinkHaLavorato();
            Iterator<Azienda> it2 = az_elem.iterator();
            while(it2.hasNext()) {
                Azienda az = it2.next();
                insieme_aziende.add(az);
            }
        }
        n = insieme_aziende.size();
        /* 2 */









        vettore_aziende = insieme_aziende.toArray();
        /* 3 */
        conta_persone = new int[n];
        /* 4 */
        it = i.iterator();
        while(it.hasNext()) {
            Persona elem = it.next();
            Set<Azienda> az_elem = elem.getLinkHaLavorato();
            Iterator<Azienda> it2 = az_elem.iterator();
            while(it2.hasNext()) {
                Azienda az = it2.next();
                conta_persone[indiceAzienda(az)]++;
            }
        }
        /* 5 */
        int max = 0;
        int indice_max = 0;
        for (int j = 0; j < n; j++)
            if (conta_persone[j] > max) {
                max = conta_persone[j];
                indice_max = j;
            }
        /* 6 */
        return (Azienda)vettore_aziende[indice_max];
    }
    private AnalisiCollocamento() { }
}
```

Soluzione esercizio 8

```
// File AssOSTAR/Esercizio/Persona.java
import java.util.*;
import IteratoreSolaLettura.*;
public class Persona {
    private final String nome;
    private HashSet<Azienda> insieme_link;
    public Persona(String n) {
        nome = n;
        insieme_link = new HashSet<Azienda>();
    }
    public String getNome() { return nome; }
    public void inserisciLinkHaLavorato(Azienda az) {
        if (az != null) insieme_link.add(az);
    }
    public void eliminaLinkHaLavorato(Azienda az) {
        if (az != null) insieme_link.remove(az);
    }
    public Iterator<Azienda> getLinkHaLavorato() {
        return new IteratoreSolaLettura<Azienda>(insieme_link.iterator());
    }
}

// File AssOSTAR/Esercizio/AnalisiCollocamento.java
```



```
import java.util.*;

public final class AnalisiCollocamento {
    private static Object[] vettore_aziende;
    private static int[] conta_persone;
    private static int n;

    private static int indiceAzienda(Azienda az) {
        // funzione di servizio
        // restituisce l'indice di az nel vettore vettore_aziende
        int result = 0;
        for (int i = 0; i < n; i++)
            if (vettore_aziende[i] == az)
                result = i;
        return result;
    }

    public static Azienda piuDipendenti (Set<Persona> i) {
        /* 1 */
        HashSet<Azienda> insieme_aziende = new HashSet<Azienda>();
        Iterator<Persona> it = i.iterator();
        while(it.hasNext()) {
            Persona elem = it.next();
            Iterator<Azienda> it2 = elem.getLinkHaLavorato();
            while(it2.hasNext()) {
                Azienda az = it2.next();
```

```
                insieme_aziende.add(az);
            }
        }
        n = insieme_aziende.size();
        /* 2 */
        vettore_aziende = insieme_aziende.toArray();
        /* 3 */
        conta_persone = new int[n];
        /* 4 */
        it = i.iterator();
        while(it.hasNext()) {
            Persona elem = it.next();
            Iterator<Azienda> it2 = elem.getLinkHaLavorato();
            while(it2.hasNext()) {
                Azienda az = it2.next();
                conta_persone[indiceAzienda(az)]++;
            }
        }
        /* 5 */
        int max = 0;
        int indice_max = 0;
        for (int j = 0; j < n; j++)
            if (conta_persone[j] > max) {
                max = conta_persone[j];
                indice_max = j;
            }









        /* 6 */
        return (Azienda)vettore_aziende[indice_max];
    }
    private AnalisiCollocamento() { }
}
```

Soluzione esercizio 9

Per l'operazione **AssunzioneInBlocco** adottiamo il seguente algoritmo:

```
per ogni Persona elem di i
    elimina, se presente, il link di tipo lavora da elem
    inserisci un link fra elem e az, con attributo an
```

Per l'operazione **AssunzionePersonaleEsperto** adottiamo il seguente algoritmo:

```
per ogni Persona elem di i
    se elem ha un link di tipo lavora con attributo annoAssunzione <= av
        allora
            elimina tale link di tipo lavora da elem
            inserisci un link fra elem e az, con attributo an
```

Gli algoritmi vengono realizzati tramite le funzioni `assunzioneInBlocco()` ed `assunzionePersonaleEsperto()` della seguente classe Java.

```
// File Ass01Attr/RistrutturazioneIndustriale.java

import java.util.*;

public final class RistrutturazioneIndustriale {
    public static void assunzioneInBlocco
        (Set<Persona> i, Azienda az, int an) {
        Iterator<Persona> it = i.iterator();
```



```
        while(it.hasNext()) {
            Persona elem = it.next();
            elem.eliminaLinkLavora();
            TipoLinkLavora temp = null;
            try {
                temp = new TipoLinkLavora(az,elem,an);
            }
            catch (EccezionePrecondizioni e) {
                System.out.println(e);
            }
            elem.inserisciLinkLavora(temp);
        }
    }
    public static void assunzionePersonaleEsperto
        (Set<Persona> i, Azienda az, int av, int an) {
        Iterator<Persona> it = i.iterator();
        while(it.hasNext()) {
            Persona elem = it.next();
            if (elem.getLinkLavora() != null &&
                elem.getLinkLavora().getAnnoAssunzione() <= av) {
                elem.eliminaLinkLavora();
                TipoLinkLavora temp = null;
                try {
                    temp = new TipoLinkLavora(az,elem,an);
                }
                catch (EccezionePrecondizioni e) {
```

```
                    System.out.println(e);
                }
                elem.inserisciLinkLavora(temp);
            }
        }
    }
    private RistrutturazioneIndustriale() { }
}
```

Soluzione esercizio 10

Per l'operazione **PeriodoPiùLungo** adottiamo il seguente algoritmo:

```
int max = 0;
per ogni link l di tipo Ha_lavorato in cui p è coinvolto
    int durata = l.annoFine - l.annoInizio + 1;
    se durata > max
        allora
            max = durata;
return max;
```

Per l'operazione **RiAssuntoSubito** adottiamo il seguente algoritmo:

```
int max = 0;
per ogni link lnk di tipo Ha_lavorato in cui p è coinvolto
    per ogni link lnk2 di tipo Ha_lavorato in cui p è coinvolto
        se lnk.annoFine == lnk2.annoInizio -1;
            allora return true;
return false;
```

Soluzione esercizio 11

Per l'operazione **SonoStatiColleghi** adottiamo il seguente algoritmo di cui viene dato il primo raffinamento; raffinamenti successivi sono lasciati per esercizio.

```
Insieme(link di tipo Ha_lavorato) lavori_p1 =
    link di tipo Ha_lavorato in cui p1 è coinvolto;
Insieme(link di tipo Ha_lavorato) lavori_p2 =
    link di tipo Ha_lavorato in cui p2 è coinvolto;
per ogni link lnk di lavori_p1
    se esiste in lavori_p2 un link compatibile con lnk
        allora return true;
return false;
```

Gli algoritmi vengono realizzati tramite le funzioni `periodoPiuLungo()`, `riAssuntoSubito()` e `sonoStatiColleghi()` della seguente classe Java. Si noti la presenza di alcune funzioni di servizio `private`.

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

public final class AnalisiMercatoLavoro {
    public static int periodoPiuLungo(Persona p) {
        int max = 0;
        Set<TipoLinkHaLavorato> temp = p.getLinkHaLavorato();
        Iterator<TipoLinkHaLavorato> it = temp.iterator();
        while(it.hasNext()) {
```



```
            TipoLinkHaLavorato lnk = it.next();
            int durata = lnk.getAnnoFine() - lnk.getAnnoInizio() + 1;
            if (durata > max)
                max = durata;
        }
        return max;
    }
    public static boolean riAssuntoSubito(Persona p) {
        Set<TipoLinkHaLavorato> temp = p.getLinkHaLavorato();
        Iterator<TipoLinkHaLavorato> it = temp.iterator();
        while(it.hasNext()) {
            TipoLinkHaLavorato lnk = it.next();
            Iterator<TipoLinkHaLavorato> it2 = temp.iterator();
            while(it2.hasNext()) {
                TipoLinkHaLavorato lnk2 = it2.next();
                if (lnk.getAnnoFine() == lnk2.getAnnoInizio() - 1)
                    return true;
            }
        }
        return false;
    }
    public static boolean sonoStatiColleghi(Persona p1, Persona p2) {
        Set<TipoLinkHaLavorato> lavori_p1 = p1.getLinkHaLavorato();
        Set<TipoLinkHaLavorato> lavori_p2 = p2.getLinkHaLavorato();
        Iterator<TipoLinkHaLavorato> it = lavori_p1.iterator();
        while(it.hasNext()) {
```

```
            TipoLinkHaLavorato lnk = it.next();
            if (contieneCompatibile(lnk,lavori_p2))
                return true;
        }
        return false;
    }
    private static boolean contieneCompatibile
        (TipoLinkHaLavorato t, Set<TipoLinkHaLavorato> ins) {
        // funzione di servizio: verifica se nell'insieme di link ins
        // sia presente un link compatibile con t
        Iterator<TipoLinkHaLavorato> it = ins.iterator();
        while(it.hasNext()) {
            TipoLinkHaLavorato lnk = 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();









    }
    private AnalisiMercatoLavoro() { }
}
```

Soluzione esercizio 12

Per l'operazione **Promuovi** adottiamo il seguente algoritmo di cui viene dato il primo raffinamento; raffinamenti successivi sono lasciati per esercizio.

```
1. calcola l'anno min della stanza che è occupata da più tempo;
2. scegli una persona p di ins tale che p.occupazione.daAnno == p;
   elimina il link di tipo occupazione per p;
   crea un nuovo link di tipo occupazione per p, con la stanza e l'anno
   specificati;
```

Per l'operazione **Libera** adottiamo il seguente algoritmo di cui viene dato il primo raffinamento; raffinamenti successivi sono lasciati per esercizio.

```
1. calcola l'anno min della stanza che è occupata da più tempo;
2. per tutte le persone p di ins tale che p.occupazione.daAnno == p
       elimina il link di tipo occupazione per p;
```


Soluzione esercizio 13

Per l'operazione **Trasloca** adottiamo il seguente algoritmo:

```
per ogni persona p1 di i1
    scegli una persona p2 di i2, non riscegliendo mai la stessa;
    elimina il link di tipo occupazione per p2;
    crea un nuovo link di tipo occupazione per p1, con la stanza e l'anno
    specificati;
```

Gli algoritmi vengono realizzati tramite le funzioni `promuovi()`, `libera()` e `trasloca()` della seguente classe Java.

```
// File RespEntrambi01/RiallocazionePersonale.java
import java.util.*;

public final class RiallocazionePersonale {
    public static void promuovi
        (Set<Persona> ins, Stanza st, int anno) {
        /* 1 */
        Iterator<Persona> it = ins.iterator();
        int min = (it.next()).getLinkOccupazione().getDaAnno();
        while(it.hasNext()) {
            Persona p = it.next();
            if (p.getLinkOccupazione().getDaAnno() < min)
                min = p.getLinkOccupazione().getDaAnno();
        }
```

```
    /* 2 */
    it = ins.iterator();
    boolean trovato = false;
    while (!trovato) {
        Persona p = it.next();
        if (p.getLinkOccupazione().getDaAnno() == min) {
            ManagerOccupazione.elimina(p.getLinkOccupazione());
            TipoLinkOccupazione t = null;
            try {
                t = new TipoLinkOccupazione(st,p,anno);
            }
            catch (EccezionePrecondizioni e) {
                System.out.println(e);
            }
            ManagerOccupazione.inserisci(t);
            trovato = true;
        }
    }
}
public static void libera(Set<Stanza> ins) {
    /* 1 */
    Iterator<Stanza> it = ins.iterator();
    int min = (it.next()).getLinkOccupazione().getDaAnno();
    while(it.hasNext()) {
        Stanza s = it.next();
        if (s.getLinkOccupazione().getDaAnno() < min)
            min = s.getLinkOccupazione().getDaAnno();
    }
    /* 2 */
    it = ins.iterator();
    while(it.hasNext()) {
        Stanza s = it.next();
        if (s.getLinkOccupazione().getDaAnno() == min) {
            ManagerOccupazione.elimina(s.getLinkOccupazione());
        }
    }
}
public static void trasloca
    (Set<Persona> ins1, Set<Persona> ins2, int anno) {
    Iterator<Persona> it1 = ins1.iterator();
    Iterator<Persona> it2 = ins2.iterator();
    while(it1.hasNext()) {
        Persona p1 = it1.next();
        Persona p2 = it2.next();
        TipoLinkOccupazione t2 = p2.getLinkOccupazione();
        Stanza st = t2.getStanza();
        TipoLinkOccupazione t1 = null;
        try {
            t1 = new TipoLinkOccupazione(st,p1,anno);
        }
        catch (EccezionePrecondizioni e) {
            System.out.println(e);
```

```
            }
            ManagerOccupazione.elimina(t2);
            ManagerOccupazione.inserisci(t1);
        }
    }
    private RiallocazionePersonale() { }
}
```

Soluzione esercizio 14

Per l'operazione **TrovaNuovi** adottiamo il seguente algoritmo:

```
Insieme(link di tipo Residenza) res = c.residenza;
Insieme(Persona) out = insieme vuoto;
per ogni link lnk di res
    se lnk.daAnno >= anno
        allora inserisci lnk.Persona in out;
return out;
```

L'algoritmo viene realizzato tramite la funzione `trovaNuovi()` della seguente classe Java.

```
// File RespEntrambiOSTAR/GestioneAnagrafe.java
import java.util.*;
public final class GestioneAnagrafe {
    public static Set<Persona> trovaNuovi(Citta c, int anno) {
        Set<TipoLinkResidenza> res = c.getLinkResidenza();
        HashSet<Persona> out = new HashSet<Persona>();
        Iterator<TipoLinkResidenza> it = res.iterator();
        while(it.hasNext()) {
            TipoLinkResidenza lnk = it.next();
            if (lnk.getDaAnno() >= anno)
                out.add(lnk.getPersona());
        }
        return out;
```

```
    }
    private GestioneAnagrafe() { }
}
```

Soluzione esercizio 15

Per l'operazione **StudenteBravo** adottiamo il seguente algoritmo:

```
per ogni link lnk di tipo Esame in cui s è coinvolto
    se lnk.voto < 27
        allora return false
return true;
```

Per l'operazione **CorsoFacile** adottiamo il seguente algoritmo:

```
per ogni link lnk di tipo Esame in cui c è coinvolto
    se lnk.voto < 27
        allora return false
return true;
```

Gli algoritmi vengono realizzati tramite le funzioni `studenteBravo()` e `corsoFacile()` della seguente classe Java.

```
// File RespEntrambiOSTAR2/ValutazioneDidattica.java

import java.util.*;

public final class ValutazioneDidattica {
    public static boolean studenteBravo(Studente s) {
        Set<TipoLinkEsame> es = s.getLinkEsame();
        Iterator<TipoLinkEsame> it = es.iterator();
```

```
        while(it.hasNext()) {
            TipoLinkEsame lnk = it.next();
            if (lnk.getVoto() < 27)
                return false;
        }
        return true;
    }
    public static boolean corsoFacile(Corso c) {
        Set<TipoLinkEsame> es = c.getLinkEsame();
        Iterator<TipoLinkEsame> it = es.iterator();
        while(it.hasNext()) {
            TipoLinkEsame lnk = it.next();
            if (lnk.getVoto() < 27)
                return false;
        }
        return true;
    }
    private ValutazioneDidattica() { }
}
```

Soluzione esercizio 16

Gli algoritmi sono identici a quelli forniti come soluzione per l'esercizio 15.

```
// File MoltMax/ValutazioneDidattica.java

import java.util.*;

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

```
        }
        return true;
    }
    private ValutazioneDidattica() { }
}
```

Soluzione esercizio 17

Per l'operazione **Cambiano** adottiamo il seguente algoritmo:

```
per ogni studente s di i
    se s non è iscritto al corso di laurea c
        allora elimina il link di tipo iscritto da s;
               crea un nuovo link di tipo iscritto per s,
               con il corso di laurea e l'anno specificati;
```

L'algoritmo viene realizzato tramite la funzione `cambiano()` della seguente classe Java.

```
// File MoltMax1/CambiamentoIndirizzo.java

import java.util.*;

public final class CambiamentoIndirizzo {
    public static void cambiano(Set<Studente> i, CorsoDiLaurea c,
                                int anno)
        throws EccezioneCardMin {
        Iterator<Studente> it = i.iterator();
        while(it.hasNext()) {
            Studente s = it.next();
            if (s.getLinkIscritto().getCorsoDiLaurea() != c) {
                ManagerIscritto.elimina(s.getLinkIscritto());
                TipoLinkIscritto t = null;
                try {
```

```
                    t = new TipoLinkIscritto(c,s,anno);
                }
                catch (EccezionePrecondizioni e) {
                    System.out.println(e);
                }
                ManagerIscritto.inserisci(t);
            }
        }
    }
    private CambiamentoIndirizzo() { }
}
```

Soluzione esercizio 18

Notiamo immediatamente che i vincoli di molteplicità impongono la responsabilità doppia su entrambe le associazioni.

Per quanto riguarda le strutture di dati per la rappresentazione delle associazioni:

- le classi Java `Docente`, `Facolta` e `Corso` avranno un campo di tipo `Insieme-ListaOmogeneo`, in quanto sono tutte coinvolte in una associazione con molteplicità massima maggiore di uno;
- la classe Java `Docente` avrà inoltre un campo dato di tipo `TipoLinkDipendente`.

Le classi Java `Docente`, `Facolta` e `Corso` avranno, per ogni associazione `ASSOC` in cui sono coinvolte:

- una funzione `quantiASSOC()`;
- la funzione `getLinkASSOC()`; tale funzione lancerà una eccezione di tipo `EccezioneCardMin` e/o `EccezioneCardMax`, a seconda dei vincoli;

Soluzione esercizio 18 (cont.)

La classe Java `AssociazioneDipendente` (coinvolta in un vincolo di molteplicità massima uguale a uno) deve effettuare i controlli del caso, in particolare:

- che l'inserimento avvenga solo se il docente non è già dipendente (sfruttando la funzione `quantiDipendente()` di `Docente`),
- che la cancellazione avvenga solo per link esistenti.

La maniera di realizzare le seguenti classi Java è quella usuale:

- `AssociazioneIscritto`,
- `TipoLinkDipendente`,
- `TipoLinkInsegna`,
- `EccezionePrecondizioni`,
- `EccezioneCardMin`,
- `EccezioneCardMax`.

Sol. eserc. 18: classe Java Facolta

```
// File MoltMax1/Esercizio/Facolta.java
import java.util.*;
public class Facolta {
    private final String nome;
    private HashSet<TipoLinkDipendente> insieme_link;
    public static final int MIN_LINK_DIPENDENTE = 8;
    public Facolta(String n) {
        nome = n;
        insieme_link = new HashSet<TipoLinkDipendente>();
    }
    public String getNome() { return nome; }
    public int quantiDipendenti() { return insieme_link.size(); }
    public void inserisciLinkDipendente(TipoLinkDipendente t) {
      if (t != null && t.getFacolta()==this)
         ManagerDipendente.inserisci(t);
    }
    public void eliminaLinkDipendente(TipoLinkDipendente t) {
      if (t != null && t.getFacolta()==this)
         ManagerDipendente.elimina(t);
    }
    public Set<TipoLinkDipendente> getLinkDipendente() throws EccezioneCardMin {
        if (quantiDipendenti() < MIN_LINK_DIPENDENTE)
            throw new EccezioneCardMin("Cardinalita' minima violata");
        else return (HashSet<TipoLinkDipendente>)insieme_link.clone();
```

```
    }
    public void inserisciPerManagerDipendente(ManagerDipendente a) {
        if (a != null) insieme_link.add(a.getLink());
    }
    public void eliminaPerManagerDipendente(ManagerDipendente a) {
        if (a != null) insieme_link.remove(a.getLink());
    }
}
```

Sol. eserc. 18: classe Java Corso

```
// File MoltMax1/Esercizio/Corso.java
import java.util.*;
public class Corso {
    private final String nome;
    private HashSet<TipoLinkInsegna> insieme_link;
    public static final int MIN_LINK_INSEGNA = 1;
    public Corso(String n) {
        nome = n;
        insieme_link = new HashSet<TipoLinkInsegna>();
    }
    public String getNome() { return nome; }
    public int quantiInsegna() { return insieme_link.size(); }
    public void inserisciLinkinsegna(TipoLinkInsegna t) {
      if (t != null && t.getCorso()==this)
         ManagerInsegna.inserisci(t);
    }
    public void eliminaLinkInsegna(TipoLinkInsegna t) {
      if (t != null && t.getCorso()==this)
         ManagerInsegna.elimina(t);
    public Set<TipoLinkInsegna> getLinkInsegna() throws EccezioneCardMin {
        if (quantiInsegna() < MIN_LINK_INSEGNA)
            throw new EccezioneCardMin("Cardinalita' minima violata");
        else return (HashSet<TipoLinkInsegna>)insieme_link.clone();
    }
```

```
    public void inserisciPerManagerInsegna(ManagerInsegna a) {
        if (a != null) insieme_link.add(a.getLink());
    }
    public void eliminaPerManagerInsegna(ManagerInsegna a) {
        if (a != null) insieme_link.remove(a.getLink());
    }

}
```

Sol. eserc. 18: classe Java Docente

```
// File MoltMax1/Esercizio/Docente.java
import java.util.*;
public class Docente {
    private final String cognome;
    private TipoLinkDipendente link_dipendente;
    private HashSet<TipoLinkInsegna> insieme_link_insegna;
    public static final int MIN_LINK_DIPENDENTE = 1;
    public static final int MIN_LINK_INSEGNA = 1;
    public static final int MAX_LINK_INSEGNA = 3;
    public Docente(String n) {
        cognome = n;
        insieme_link_insegna = new HashSet<TipoLinkInsegna>();
    }
    public String getCognome() { return cognome; }
    public int quantiDipendente() {
        if (link_dipendente == null)
            return 0;
        else return 1;
    }
    public void inserisciLinkDipendente(TipoLinkDipendente t) {
      if (t != null && t.getDocente()==this)
         ManagerDipendente.inserisci(t);
    }
    public void eliminaLinkDipendente(TipoLinkDipendente t) {
```

```
      if (t != null && t.getDocente()==this)
         ManagerDipendente.elimina(t);
    }
    public TipoLinkDipendente getLinkDipendente() throws EccezioneCardMin {
        if (link_dipendente == null)
            throw new EccezioneCardMin("Cardinalita' minima violata");
        else
            return link_dipendente;
    }
    public int quantiInsegna() {
        return insieme_link_insegna.size();
    }
    public void inserisciLinkInsegna(TipoLinkInsegna t) {
      if (t != null && t.getDocente()==this)
         ManagerInsegna.inserisci(t);
    }
    public void eliminaLinkInsegna(TipoLinkInsegna t) {
      if (t != null && t.getDocente()==this)
         ManagerInsegna.elimina(t);
    }
    public Set<TipoLinkInsegna> getLinkInsegna() throws EccezioneCardMin,
                                                        EccezioneCardMax {
        if (quantiInsegna() < MIN_LINK_INSEGNA)
            throw new EccezioneCardMin("Cardinalita' minima violata");
        if (quantiInsegna() > MAX_LINK_INSEGNA)
            throw new EccezioneCardMax("Cardinalita' massima violata");
        else return (HashSet<TipoLinkInsegna>)insieme_link_insegna.clone();
    }
    public void inserisciPerManagerDipendente(ManagerDipendente a) {
        if (a != null) link_dipendente = a.getLink();
    }
    public void eliminaPerManagerDipendente(ManagerDipendente a) {
        if (a != null) link_dipendente = null;
    }
    public void inserisciPerManagerInsegna(ManagerInsegna a) {
        if (a != null) insieme_link_insegna.add(a.getLink());
    }
    public void eliminaPerManagerInsegna(ManagerInsegna a) {
        if (a != null) insieme_link_insegna.remove(a.getLink());
    }
}
```

Sol. eserc. 18: classe AssociazioneDipendente

Sol. eserc. 19: classe Java Studente

```
// File Ternaria/Studente.java
import java.util.*;
public class Studente {
    private final String nome, cognome, matricola;
    private int eta;
    private HashSet<TipoLinkEsame> insieme_link;
    public Studente(String n, String c, String m, int e) {
        nome = n;
        cognome = c;
        matricola = n;
        eta = e;
        insieme_link = new HashSet<TipoLinkEsame>();
    }
    public String getMatricola() { return matricola; }
    public String getNome() { return nome; }
    public String getCognome() { return cognome; }
    public int getEta() { return eta; }
    public void setEta(int e) { eta = e; }
    public void inserisciLinkEsame(TipoLinkEsame t) {
      if (t != null && t.getStudente()==this)
         ManagerEsame.inserisci(t);
    }
    public void eliminaLinkEsame(TipoLinkEsame t) {
      if (t != null && t.getStudente()==this)
```

```
        ManagerEsame.elimina(t);
    }
    public Set<TipoLinkEsame> getLinkEsame() {
        return (HashSet<TipoLinkEsame>)insieme_link.clone();
    }
    public void inserisciPerManagerEsame(ManagerEsame a) {
        if (a != null) insieme_link.add(a.getLink());
    }
    public void eliminaPerManagerEsame(ManagerEsame a) {
        if (a != null) insieme_link.remove(a.getLink());
    }
}
```

Sol. eserc. 19: classe Java Professore

```
// File Ternaria/Professore.java
import java.util.*;
public class Professore {
    private final String codice;
    private int eta;
    private HashSet<TipoLinkEsame> insieme_link;
    public Professore(String c, int e) {
        codice = c;
        eta = e;
        insieme_link = new HashSet<TipoLinkEsame>();
    }
    public String getCodice() { return codice; }
    public int getEta() { return eta; }
    public void setEta(int e) { eta = e; }
    public void inserisciLinkEsame(TipoLinkEsame t) {
      if (t != null && t.getProfessore()==this)
        ManagerEsame.inserisci(t);
    }
    public void eliminaLinkEsame(TipoLinkEsame t) {
      if (t != null && t.getProfessore()==this)
        ManagerEsame.elimina(t);
    }
    public Set<TipoLinkEsame> getLinkEsame() {
        return (HashSet<TipoLinkEsame>)insieme_link.clone();
```

```
    }
    public void inserisciPerManagerEsame(ManagerEsame a) {
        if (a != null) insieme_link.add(a.getLink());
    }
    public void eliminaPerManagerEsame(ManagerEsame a) {
        if (a != null) insieme_link.remove(a.getLink());
    }
}
```

Sol. eserc. 19: classe Java Corso

```
// File Ternaria/Corso.java
public class Corso {
    private final String nome, disciplina;
    public Corso(String n, String d) {
        nome = n;
        disciplina = d;
    }
    public String getNome() { return nome; }
    public String getDisciplina() { return disciplina; }
}
```

Sol. eserc. 19: classe Java AssociazioneEsame

Sol. eserc. 19: classe Java TipoLinkEsame

```
// File Ternaria/TipoLinkEsame.java
public class TipoLinkEsame {
    private final Corso ilCorso;
    private final Professore ilProfessore;
    private final Studente loStudente;
    private final int voto;
    public TipoLinkEsame(Corso x, Studente y, Professore z, int a)
        throws EccezionePrecondizioni {
        if (x == null || y == null || z == null)  // CONTROLLO PRECONDIZIONI
            throw new EccezionePrecondizioni
                ("Gli oggetti devono essere inizializzati");
        ilCorso = x; loStudente = y; ilProfessore = z; voto = a;
    }
    public boolean equals(Object o) {
        if (o != null && getClass().equals(o.getClass())) {
            TipoLinkEsame b = (TipoLinkEsame)o;
            return b.ilCorso == ilCorso && b.loStudente == loStudente &&
                b.ilProfessore == ilProfessore;
        }
        else return false;
    }
    public int hashCode() {
        return ilCorso.hashCode() + loStudente.hashCode() +
            ilProfessore.hashCode();
```

```
    }
    public Corso getCorso() { return ilCorso; }
    public Studente getStudente() { return loStudente; }
    public Professore getProfessore() { return ilProfessore; }
    public int getVoto() { return voto; }
}
```

Sol. eserc. 19: cliente

```
// File Ternaria/ValutazioneDidattica.java
import java.util.*;
public final class ValutazioneDidattica {
    public static boolean studenteBravo(Studente s) {
        Set<TipoLinkEsame> es = s.getLinkEsame();
        Iterator<TipoLinkEsame> it = es.iterator();
        while(it.hasNext()) {
            TipoLinkEsame lnk = it.next();
            if (lnk.getVoto() < 27)
                return false;
        }
        return true;
    }
    public static boolean professoreBuono(Professore p) {
        Set<TipoLinkEsame> es = p.getLinkEsame();
        Iterator<TipoLinkEsame> it = es.iterator();
        while(it.hasNext()) {
            TipoLinkEsame lnk = it.next();
            if (lnk.getVoto() < 27)
                return false;
        }
        return true;
    }
    private ValutazioneDidattica() { }
}
```

Sol. eserc. 20: classe Java Azienda

```
// File Ternaria/Esercizio/Azienda.java
import java.util.*;

public class Azienda {
    private final String nome;
    private final int annoFondazione;
    private HashSet<TipoLinkContenzioso> linkAccusatrice;
    private HashSet<TipoLinkContenzioso> linkAccusata;
    public Azienda(String no, int an) {
        nome = no;
        annoFondazione = an;
        linkAccusatrice = new HashSet<TipoLinkContenzioso>();
        linkAccusata= new HashSet<TipoLinkContenzioso>();
    }
    public String getNome() { return nome; }
    public int getAnnoFondazione() { return annoFondazione; }
    public void inserisciLinkAccusatrice(TipoLinkContenzioso t) {
      if (t != null && t.getAccusatrice()==this)
         ManagerContenzioso.inserisci(t);
    }
    public void eliminaLinkAccusatrice(TipoLinkContenzioso t) {
      if (t != null && t.getAccusatrice()==this)
         ManagerContenzioso.elimina(t);
    }
```



```
    public Set<TipoLinkContenzioso> getAccusatrice() {
      return (HashSet<TipoLinkContenzioso>)linkAccusatrice.clone();
    }
    public void inserisciLinkAccusata(TipoLinkContenzioso t) {
      if (t != null && t.getAccusata()==this)
         ManagerContenzioso.inserisci(t);
    }
    public void eliminaLinkAccusata(TipoLinkContenzioso t) {
      if (t != null && t.getAccusata()==this)
         ManagerContenzioso.elimina(t);
    }
    public Set<TipoLinkContenzioso> getAccusata() {
      return (HashSet<TipoLinkContenzioso>)linkAccusata.clone();
    }
    public void inserisciPerManagerAccusatrice(ManagerContenzioso a) {
      if (a != null) { linkAccusatrice.add(a.getLink()); }
    }
    public void eliminaPerManagerAccusatrice(ManagerContenzioso a) {
      if (a != null) { linkAccusatrice.remove(a.getLink()); }
    }
    public void inserisciPerManagerAccusata(ManagerContenzioso a) {
      if (a != null) { linkAccusata.add(a.getLink()); }
    }
    public void eliminaLinkAccusata(ManagerContenzioso a) {
      if (a != null) { linkAccusata.remove(a.getLink()); }
    }
```

```
    public String toString() {
        return nome + " " + annoFondazione;
    }
}
```

Sol. eserc. 20: classe Java Persona

```
// File Ternaria/Esercizio/Persona.java
import java.util.*;

public class Persona {
    private final String nome, cognome, codiceFiscale;
    private HashSet<TipoLinkContenzioso> linkContenzioso;
    public Persona(String no, String co, String cf) {
        nome = no;
        cognome = co;
        codiceFiscale = cf;
        linkContenzioso = new HashSet<TipoLinkContenzioso>();
    }
    public String getNome() { return nome; }
    public String getCognome() { return cognome; }
    public String getCodiceFiscale() { return codiceFiscale; }
    public void inserisciLinkContenzioso(TipoLinkContenzioso t) {
      if (t != null && t.getGiudice()==this)
         ManagerContenzioso.inserisci(t);
    }
    public void eliminaLinkContenzioso(TipoLinkContenzioso t) {
      if (t != null && t.getGiudice()==this)
         ManagerContenzioso.elimina(t);
    }
    public Set<TipoLinkContenzioso> getContenzioso() {
```

```
        return (HashSet<TipoLinkContenzioso>)linkContenzioso.clone();
    }
    public void inserisciPerManagerContenzioso(ManagerContenzioso a) {
        if (a != null) { linkContenzioso.add(a.getLink()); }
    }
    public void eliminaPerManagerContenzioso(ManagerContenzioso a) {
        if (a != null) { linkContenzioso.remove(a.getLink()); }
    }
    public String toString() {
        return nome + " " + cognome + " " + codiceFiscale;
    }
}
```

Sol. eserc. 20: classe Java TipoLinkContenzios

```
// File Ternaria/Esercizio/TipoLinkContenzioso.java

public class TipoLinkContenzioso {
    private final Azienda laAccusatrice;
    private final Azienda laAccusata;
    private final Persona ilGiudice;
    private final int annoInizio;
    public TipoLinkContenzioso(Azienda a, Azienda t, Persona g, int anno)
        throws EccezionePrecondizioni {
        if (a == null || t == null || g == null)  // CONTROLLO PRECONDIZIONI
            throw new EccezionePrecondizioni
                ("Gli oggetti devono essere inizializzati");
        laAccusatrice = a; laAccusata = t; ilGiudice = g; annoInizio = anno;
    }
    public boolean equals(Object o) {
        if (o != null && getClass().equals(o.getClass())) {
            TipoLinkContenzioso b = (TipoLinkContenzioso)o;
            return b.laAccusatrice == laAccusatrice &&
                   b.laAccusata == laAccusata && b.ilGiudice == ilGiudice;
        }
        else return false;
    }
    public int hashCode() {
        return laAccusatrice.hashCode() + ilGiudice.hashCode() +
```

```
            laAccusata.hashCode();
    }
    public Azienda getAccusatrice() { return laAccusatrice; }
    public Azienda getAccusata() { return laAccusata; }
    public Persona getGiudice() { return ilGiudice; }
    public int getAnnoInizio() { return annoInizio; }
    public String toString() {
        return laAccusatrice + " " + laAccusata + " " + ilGiudice + " " + annoInizio;
    }
}
```

Sol. eserc. 20: AssociazioneContenzioso

Sol. eserc. 21: Specifica del cliente

InizioSpecificaCliente **ControlloContenziosi**

DueContenziosi *(S: Insieme(Azienda)): Insieme(Azienda)*
pre: true // nessuna
post: *result* $= \{a \mid a \in S \wedge \exists a1, p1, p2\ a1 \in Azienda \wedge p1 \in Persona \wedge p2 \in Persona \wedge p1 \neq p2 \wedge \langle a, a1, p1\rangle \in Contenzioso \wedge \langle a, a1, p2\rangle \in Contenzioso\}$
// *result* è il sottoinsieme di *S* formato da tutte le aziende che sono accusatrici in almeno due contenziosi con la stessa azienda accusata

Accusata *(A: Azienda, P: Persona): booleano*
pre: true // nessuna
post: *result* = true sse $\exists a1\ a1 \in Azienda \wedge \langle a1, A, P\rangle \in Contenzioso$
// *result* è true se *A* è accusata in almeno un contenzioso in cui *P* opera come giudice

FineSpecifica

Sol. eserc. 21: Algor. per le oper. del cliente

Per l'operazione *DueContenziosi(S)* adottiamo il seguente algoritmo:

```
Insieme(Azienda) ins = insieme vuoto;
per ogni azienda a di S
    Insieme(Link di tipo Contenzioso) s1 = i link di tipo Contenzioso che
                                           coinvolgono a come accusatrice
    Insieme(Azienda) s2 = insieme vuoto;
    per ogni t di s1
         se l'azienda accusata di t è contenuta in s2
         allora inserisci a in ins
         altrimenti inserisci l'azienda accusata di t in s2
return ins;
```

Per l'operazione *Accusata(A,P)* adottiamo il seguente algoritmo:

```
Insieme(Link di tipo Contenzioso) c = i link di tipo Contenzioso che
                                      coinvolgono A come accusatrice
per ogni elemento e di c
    se (la persona che giudica in e è P)
    allora return true;
return false;
```

Sol. eserc. 21: Realizzazione del cliente

```
// File Ternaria/Esercizio/ControlloContenziosi.java

import java.util.*;

public final class ControlloContenziosi {
    private ControlloContenziosi() { }
    public static Set<Azienda> DueContenziosi(Set<Azienda> i) {
        Set<Azienda> ins = new HashSet<Azienda>();
        Iterator<Azienda> it = i.iterator();
        while(it.hasNext()) {
            Azienda a = it.next();
            Set<TipoLinkContenzioso> s1 = a.getAccusatrice();
            Set<Azienda> s2 = new HashSet<Azienda>();
            Iterator<TipoLinkContenzioso> j = s1.iterator();
            while (j.hasNext()) {
                 TipoLinkContenzioso t = j.next();
                 if (s2.contains(t.getAccusata()))
                     ins.add(a);
                 else s2.add(t.getAccusata());
            }
        }
        return ins;
    }
    public static boolean Accusata(Azienda a, Persona p) {
```



```
        Set<TipoLinkContenzioso> s = a.getAccusata();
        Iterator<TipoLinkContenzioso> it = s.iterator();
        while(it.hasNext()) {
            TipoLinkContenzioso c = it.next();
            if (c.getGiudice() == p) return true;
        }
        return false;
    }
}
```

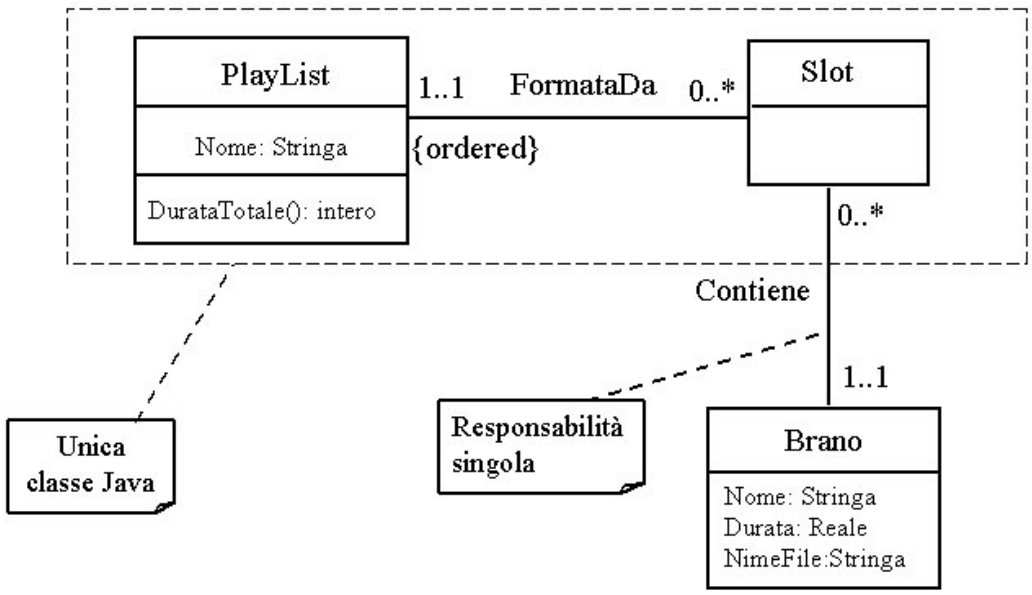
Soluzione esercizio 22

Con le precisazioni viste, possiamo fornire una realizzazione del diagramma delle classi semplificata rispetto alla metodologia fin qui presentata.

In particolare:

- poiché nella nostra applicazione non abbiamo mai bisogno di riferirci ad oggetti *Slot*, e poiché uno slot corrisponde esattamente ad una playlist, è possibile realizzare mediante un'unica classe Java `PlayList` entrambe le classi UML *PlayList* e *Slot*;
- in questa maniera, "trasferiamo" la responsabilità sull'associazione *Contiene* da *Slot* a *PlayList*;
- la classe `PlayList` avrà un campo dato di tipo `LinkedList`, per rappresentare la struttura di dati ordinata;
- possiamo eliminare una o tutte le occorrenze di un brano da una playlist;
- le realizzazione della classe Java `Brano` e dello cliente sono identiche al caso della responsabilità singola.

Soluzione esercizio 22 (cont.)



Sol. eserc. 22: classe Java PlayList

```
// File OrdinateRipetizioneOSTAR/PlayList.java
import java.util.*;
public class PlayList {
  private final String nome;
  private LinkedList<Brano> sequenza_link;
  public PlayList(String n) {
    nome = n;
    sequenza_link = new LinkedList<Brano>();
  }
  public String getNome() { return nome; }
  public void inserisciLinkContiene(Brano b) {
    if (b != null) sequenza_link.add(b);
  }
  public void eliminaPrimaOccorrenzaLinkContiene(Brano b) {
    if (b != null) sequenza_link.remove(b);
  }
  public void eliminaOgniOccorrenzaLinkContiene(Brano b) {
    if (b != null) {
      while(sequenza_link.contains(b))
        sequenza_link.remove(b);
    }
  }
  public List<Brano> getLinkContiene() {
    return (LinkedList<Brano>)sequenza_link.clone();
```



```
  }
  public int durataTotale() {
    int result = 0;
    Iterator<Brano> ib = sequenza_link.iterator();
    while (ib.hasNext()) {
      Brano b = ib.next();
      result = result + b.getDurata();
    }
    return result;
  }
}
```

Soluzione esercizio 23

Si lascia il dettaglio della fase di progetto per esercizio.

Notiamo che la responsabilità sull'associazione *Iscrizione* è doppia, mentre su *Residenza* è solo di *Persona*.

La struttura dei file e dei package è la seguente:

```
+---PackageUniversita
|   |   Citta.java
|   |   Universita.java
|   |   TipoLinkIscrizione.java
|   |   EccezionePrecondizioni.java
|   |   AssociazioneIscrizione.java
|   |   EccezioneCardMin.java
|   |   StatisticheIscrizioni.java
|   |
|   +---Persona
|   |       Persona.java
|   |
|   \---Studente
|           Studente.java
```


Sol. eserc. 23: classe Java Universita

```
// File PackageUniversita/Universita.java

package PackageUniversita;
import java.util.*;

public class Universita {
    private final String nome;
    private HashSet<TipoLinkIscrizione> insieme_link;
    public Universita(String n) {
        nome = n;
        insieme_link = new HashSet<TipoLinkIscrizione>();
    }
    public String getNome() { return nome; }
    public void inserisciLinkIscritto(TipoLinkIscrizione t) {
      if (t != null && t.getUniversita()==this)
         ManagerIscrizione.inserisci(t);
    }
    public void eliminaLinkIscritto(TipoLinkIscrizione t) {
      if (t != null && t.getUniversita()==this)
         ManagerIscrizione.elimina(t);
    }
    public Set<TipoLinkIscrizione> getLinkIscrizione() {
        return (HashSet<TipoLinkIscrizione>)insieme_link.clone();
    }
```

```
    public void inserisciPerManagerIscrizione(ManagerIscrizione a) {
        if (a != null) insieme_link.add(a.getLink());
    }
    public void eliminaPerManagerIscrizione(ManagerIscrizione a) {
        if (a != null) insieme_link.remove(a.getLink());
    }
}
```

Sol. eserc. 23: classe Java Citta

```
// File PackageUniversita/Citta.java

package PackageUniversita;

public class Citta {
    private final String nome;
    public Citta(String n) {
        nome = n;
    }
    public String getNome() { return nome; }
}
```

Sol. eserc. 23: classe Java Persona

```
// File PackageUniversita/Persona/Persona.java

package PackageUniversita.Persona;
import PackageUniversita.*;

public class Persona {
    protected final String nome, cognome;
    protected Citta residenza;
    public static final int MIN_LINK_RESIDENZA = 1;
    public Persona(String n, String c) {
        nome = n;
        cognome = c;
    }
    public String getNome() { return nome; }
    public String getCognome() { return cognome; }
    public int quantiResidenza() {
        if (residenza == null)
            return 0;
        else return 1;
    }
    public Citta getResidenza() throws EccezioneCardMin {
        if (residenza == null)
            throw new EccezioneCardMin("Cardinalita' minima violata");
        else
```



```
            return residenza;
    }
    public void setResidenza(Citta c) {
        residenza = c;
    }
}
```

Sol. eserc. 23: classe Java Studente

```
// File PackageUniversita/Studente/Studente.java

package PackageUniversita.Studente;
import PackageUniversita.*;
import PackageUniversita.Persona.*;

public class Studente extends Persona {
    protected final String matricola;
    protected TipoLinkIscrizione iscrizione;
    public static final int MIN_LINK_ISCRIZIONE = 1;
    public Studente(String n, String c, String m) {
        super(n,m);
        matricola = m;
    }
    public String getMatricola() { return matricola; }
    public int quantiIscrizione() {
        if (iscrizione == null)
            return 0;
        else return 1;
    }
    public void inserisciLinkIscrizione(TipoLinkIscrizione t) {
      if (t != null && t.getStudente()==this)
         ManagerIscrizione.inserisci(t);
    }
```



```
    public void eliminaLinkIscrizione(TipoLinkIscrizione t) {
      if (t != null && t.getStudente()==this)
         ManagerIscrizione.elimina(t);
    }
    public TipoLinkIscrizione getLinkIscrizione() throws EccezioneCardMin {
        if (iscrizione == null)
            throw new EccezioneCardMin("Cardinalita' minima violata");
        else
            return iscrizione;
    }
    public void inserisciPerManagerIscrizione(ManagerIscrizione a) {
        if (a != null) iscrizione = a.getLink();
    }
    public void eliminaPerManagerIscrizione(ManagerIscrizione a) {
        if (a != null) iscrizione = null;
    }
}
```

Sol. eserc. 23: classe Java AssociazioneIscrizio

Sol. eserc. 23: classe Java StatisticheIscrizioni

```
// File PackageUniversita/StatisticheIscrizioni.java
package PackageUniversita;
import PackageUniversita.Studente.*;
import java.util.*;
public final class StatisticheIscrizioni {
    public static Citta cittaDiProvenienza(Universita u) {
/*
LASCIATA PER ESERCIZIO. POSSIBILE ALGORITMO:
1. calcola il numero n di citta distinte in cui almeno uno fra gli studenti
   iscritti ad u è residente
2. crea un vettore vettore_citta di n elementi in cui inserisci le n
   città distinte
3. crea un vettore conta_studenti di n interi inizializzati a 0
4. memorizza in conta_studenti quanti studenti sono residenti in ciascuna città,
   utilizzando l'indice di vettore_citta
5. trova in conta_studenti l'indice indice_max dell'elemento che contiene
   il massimo valore
6. return vettore_citta[indice_max]
*/
        return null;
    }
    private StatisticheIscrizioni() { }
}
```