Progettazione del Software

La fase di realizzazione

Domenico Fabio Savo

Dipartimento di Ingegneria Informatica, Automatica e Gestionale Antonio Ruberti

Sapienza Università di Roma

Le slide di questo corso sono il frutto di una rielaborazione di analogo materiale redatto da Marco Cadoli, Giuseppe De Giacomo, Maurizio Lenzerini e Domenico Lembo

La fase di realizzazione

La fase di realizzazione si occupa di:

- scrivere il **codice** del programma, e
- produrre parte della **documentazione**:
  - struttura dei **file e dei package**.

Il suo input è costituito da:

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

La fase di realizzazione (cont.)

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

- Traduzione in Java del diagramma degli use case.
- Traduzione in Java del diagramma delle classi.

Nota: per pure esigenze di compattezza del codice, nel seguito in alcuni esempi ed esercizi adotteremo per la verifica di eventuali precondizioni l'approccio lato client, mentre si lascia per esercizio l'approccio alla verifica lato server (in genere preferibile)

Traduzione in Java del diagramma degli use case

Per semplicità, come fatto nella fase di anailisi e di progetto, non considereremo diagrammi degli use case con inclusioni, estensioni o generalizzazioni.

Uno use case *U* si realizza in Java nel seguente modo:

- una classe Java `public final U`, tipicamente da sola in un file `U.java`;
- il costruttore di `U` è privato;
- una funzione `public static` di `U` per ogni operazione di *U*.

Non siamo interessati infatti ad avere oggetti di `U`: la loro creazione è inibita rendendo privato il costruttore.

Non ha senso pensare a sottoclassi di `U`: la loro creazione è inibita dichiarando la classe `final` .

Questa classe è un mero **contenitore di funzioni**.

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 classi UML con soli attributi

Assumiamo, per ora, 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 classi UML con soli attributi

- 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** e si dichiarano `final`.

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`.
- come ogni classe `C` è derivata da `Object` (senza uso di `extends`).

Metodologia: i 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 un riferimento ad un oggetto nulla impedisce di modificare l'oggetto stesso. Quindi l'efficacia di* **`final`** *è limitata.*

Metodologia: i campi funzione

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 oggetti sono uguali solo se in realtà sono lo stesso oggetto e quindi il comportamento di default della funzione `equals()` è corretto.

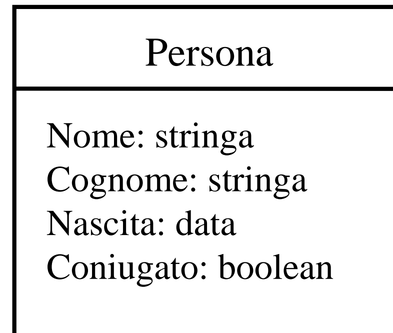
`clone()`: in molti casi, è ragionevole decidere di **non mettere a disposizione la possibilità di copiare un oggetto**, e non rendere disponibile la funzione `clone()` (non facendo overriding della funzione `protected` ereditata da `Object`).

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

`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

Classe UML	**Proprietà** nota alla nascita	**Proprietà** 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;
  } // si noti che coniugato è posto a false per default
  public String getNome() {
    return nome;
```

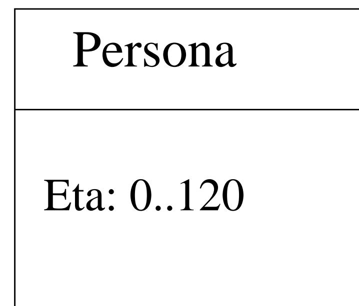
```
}
public String getCognome() {
  return cognome;
}
public int getGiornoNascita() {
  return giorno_nascita;
}
public int getMeseNascita() {
  return mese_nascita;
}
public int getAnnoNascita() {
  return anno_nascita;
}
public void setConiugato(boolean c) {
  coniugato = c;
}
public boolean getConiugato()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

Vedremo ora il codice della classe `Persona` con verifica lato server.

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("L'eta' deve essere compresa fra 0 e 120");
    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`.

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 dati 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`:

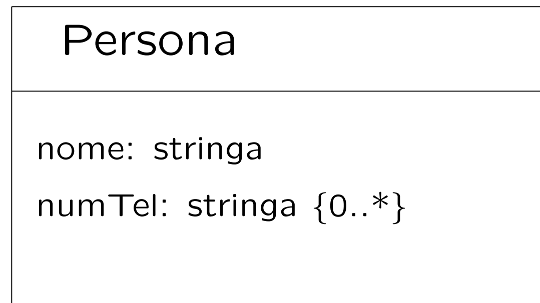
- per la scrittura del campo sono necessarie due funzioni, rispettivamente per l'inserimento di elementi nell'insieme e per la loro cancellazione;

- 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 dati 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

Soluzione esercizio 1 (1/2)

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;
```

Soluzione esercizio 1 (2/2)

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() {};
}
```

Realizzare classi con attributi e operazioni

- 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.

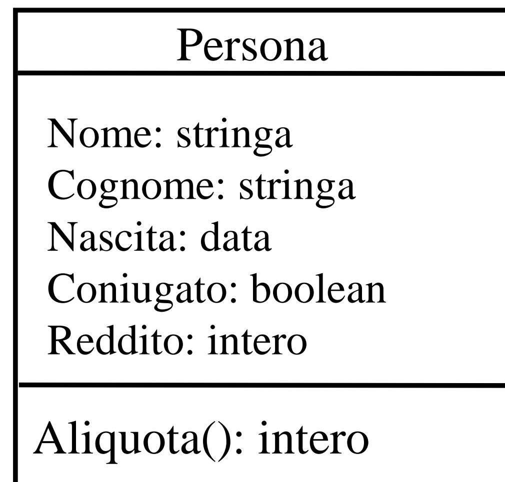
- Ogni operazione viene realizzata da una funzione **`public`** della classe Java.

  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 2: 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

Soluzione esercizio 2

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

Esercizio 3: 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

Soluzione esercizio 3 (1/2)

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.

Soluzione esercizio 3 (2/2)

```
// 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() {}
}
```

Realizzazione di associazioni

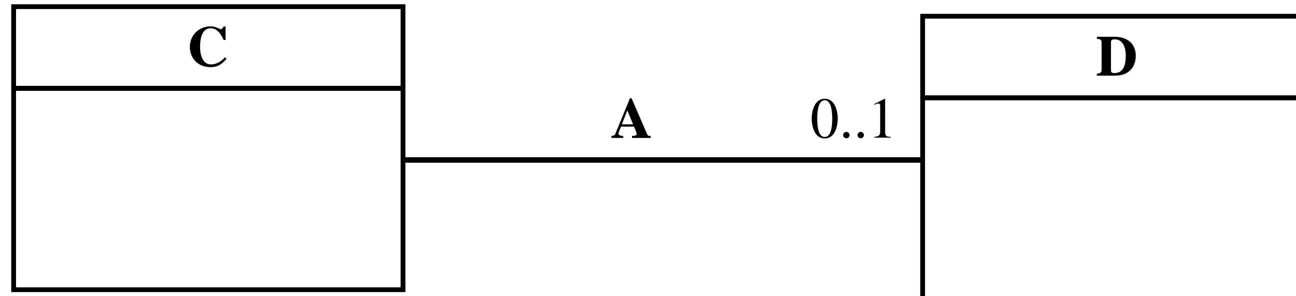
Nell'esposizione di questo argomento, seguiremo quest'ordine:

- 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.

Realizzazione di associazioni (NOTA)

Nota: per pure esigenze di compattezza del codice, nel seguito in alcuni esempi ed esercizi adotteremo per la verifica di eventuali precondizioni l'approccio lato client, mentre si lascia per esercizio l'approccio alla verifica lato server (in genere preferibile)

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



Consideriamo il caso in cui

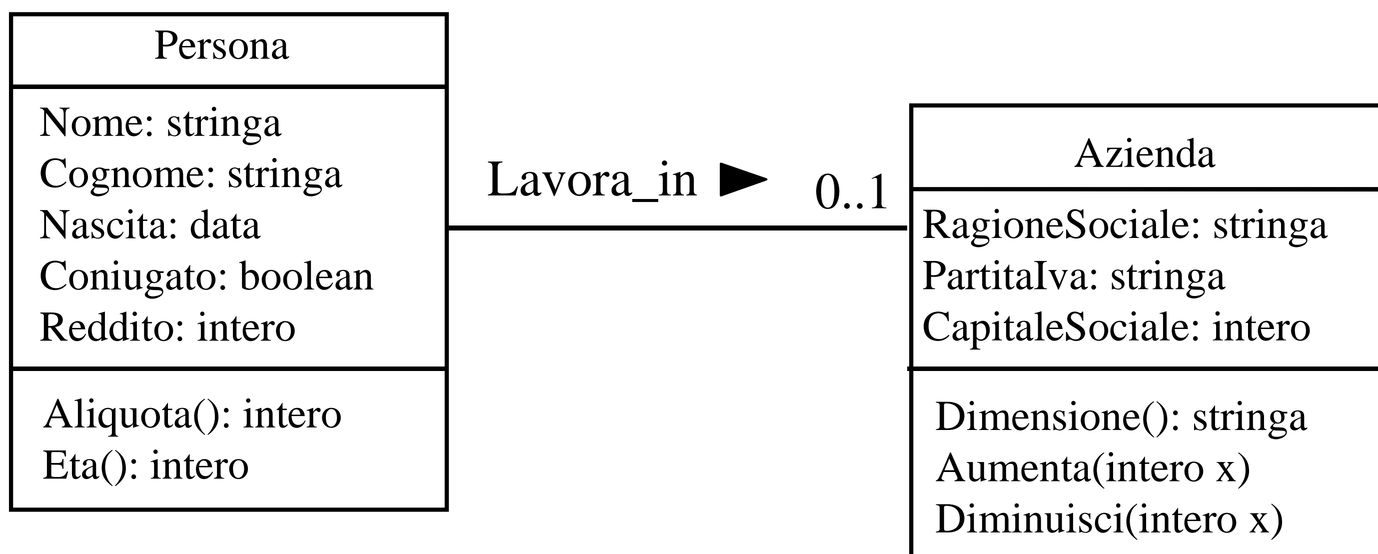
- l'associazione sia binaria;
- l'associazione colleghi ogni istanza di C a zero o una istanza di D (molteplicità 0..1),
- la *tabella delle responsabilità* prodotta in fase di progetto ci dica che C è l'unica ad avere responsabilità sull'associazione A (cioè dobbiamo realizzare un "solo verso" della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

Associazione con molteplicità 0..1 a responsabilità singola e 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 dati 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**;
- il capitale sociale viane modificato solo attraverso le funzioni `Aumenta(int)` e `Diminuisci(int)`;
- 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 4: 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

Soluzione esercizio 4 (1/2)

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;
```

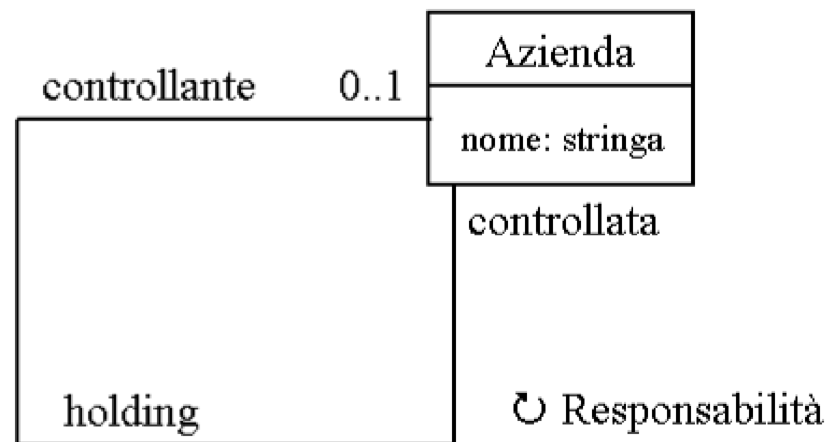
L'algoritmo viene realizzato tramite la funzione `redditoMedioInGrandiAziende()` della seguente classe Java.

Soluzione esercizio 4 (2/2)

```
// 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() { }
}
```

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.



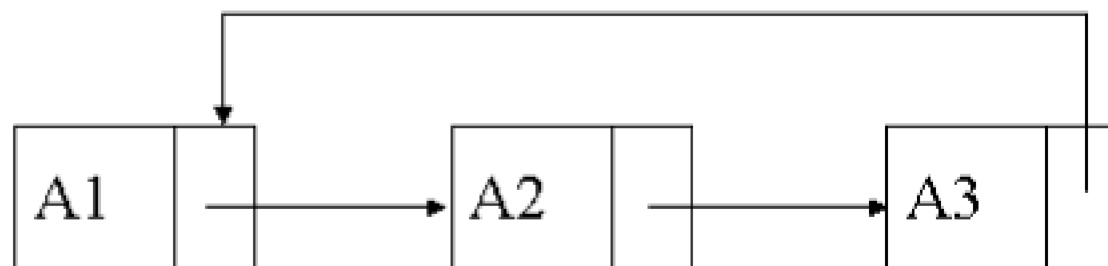
Supponiamo che la classe *Azienda* abbia la responsabilità su *holding*, solo nel ruolo *controllata*. Questo significa che, dato un oggetto x della classe *Azienda*, vogliamo poter eseguire operazioni su x per conoscere l'azienda controllante, per aggiornare l'azienda controllante, ecc.

Associazioni che insistono più volte sulla stessa classe: esempio

In questo caso, il nome del campo dati 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 5: 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

Soluzione esercizio 5 (1/2)

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

```
boolean controllataDaSeStessa(Azienda a)
  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()`.

Soluzione esercizio 5 (2/2)

```
// 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() { }
}
```

Associazioni con molteplicità 0..* a responsabilità singola e 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.

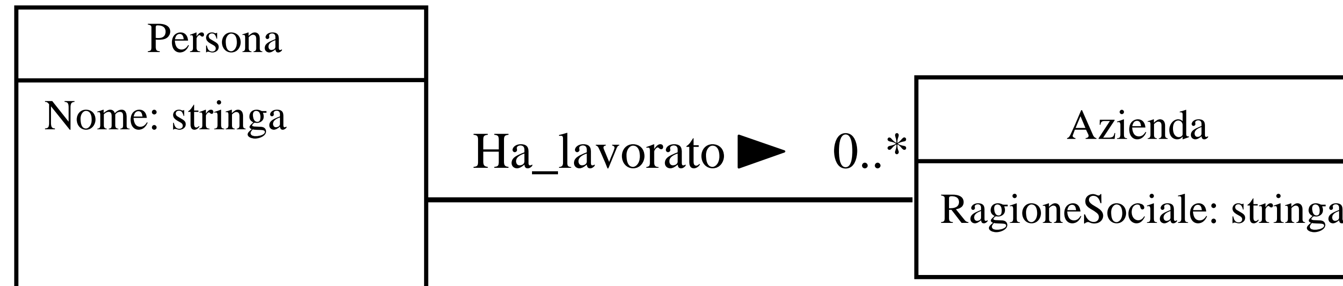
Associazioni con molteplicità 0..* a responsabilità singola e senza attributi (cont.)

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 dati 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 e senza attributi: esempio



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 dati 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 *HaLavorato* vengano gestiti solamente dalla classe `Persona`, che ha responsabilità sull'associazione.

- Per semplicità, nel seguito utilizzeremo sempre lo schema realizzativo senza condivisione di memoria.

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.

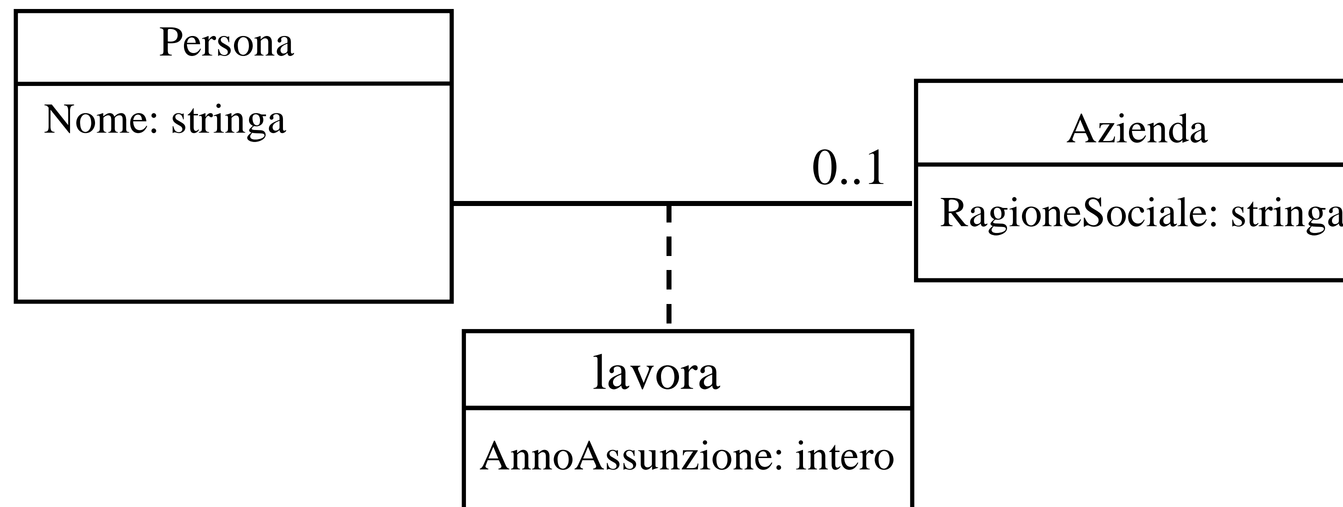
Considereremo

- 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, scegliendo 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 ass. 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 dati 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 dati 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 *(per essere precisi tali riferimenti sono variabili che contengono 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:

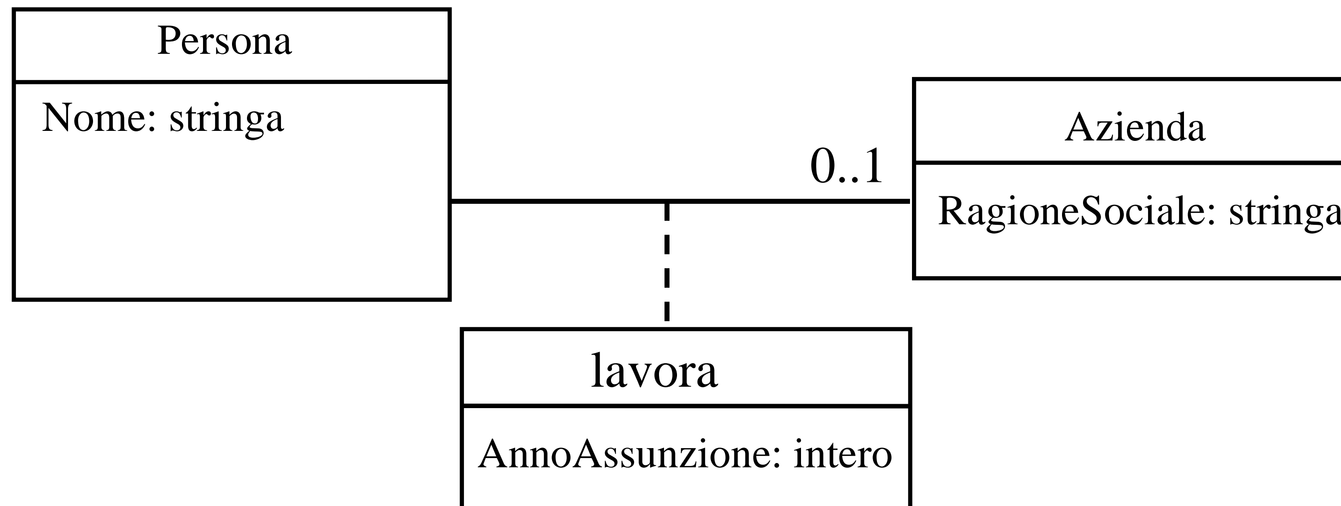
- un campo dati di tipo `TipoLinkA`, per rappresentare l'eventuale link;

  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:

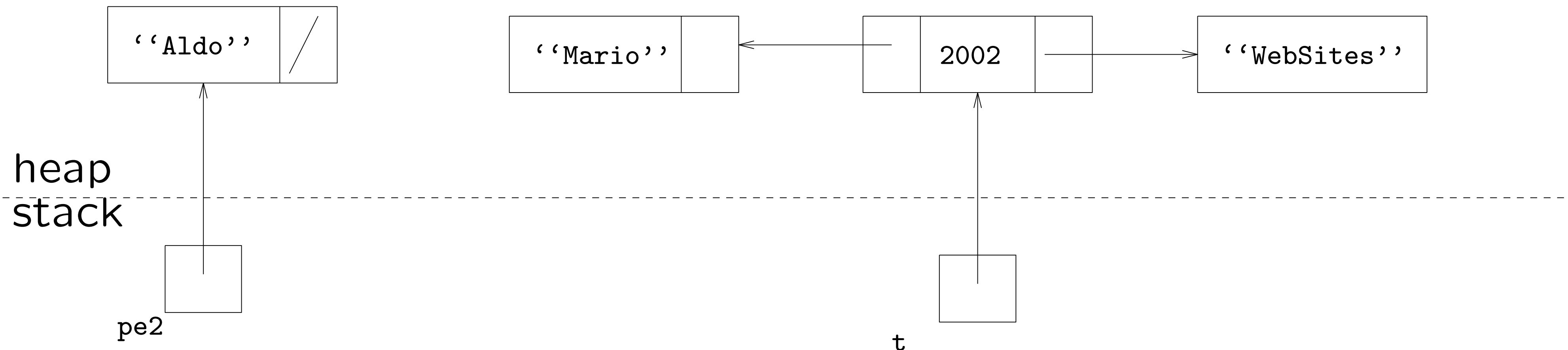
  – la persona oggetto di invocazione non sia già associata ad un link;

  – 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

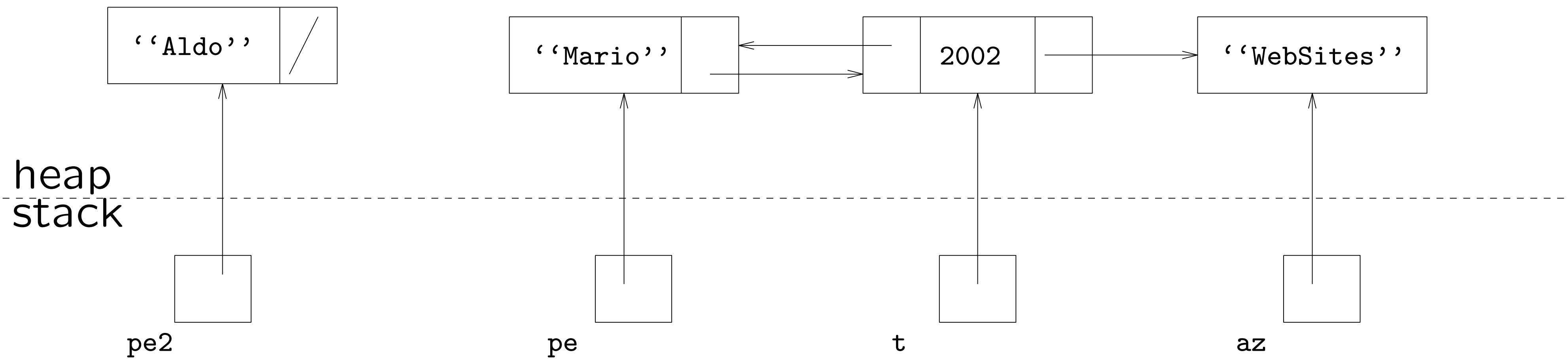


Il link `t` non si riferisce all'oggetto "Aldo".

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");
Persona 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 6: 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

Soluzione esercizio 6 (1/2)

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
```

Soluzione esercizio 6 (2/2)

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

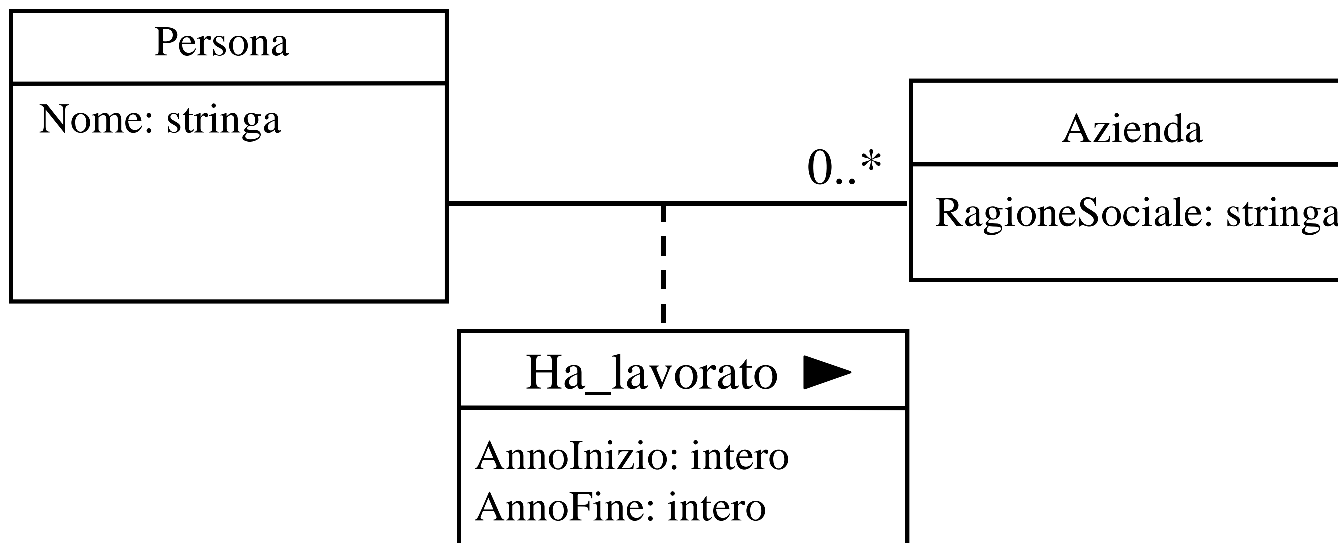
```
// File Ass01Attr/RistrutturazioneIndustriale.java

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() { }
}
```

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`);

   Ricordiamo che ogni volta che realizziamo una classe tipoLink, 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 7: 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

Soluzione esercizio 7 (1/4)

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 7 (2/4)

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;
```

Soluzione esercizio 7 (3/4)

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() { }
}
```

Esercizio 7 (4/4)

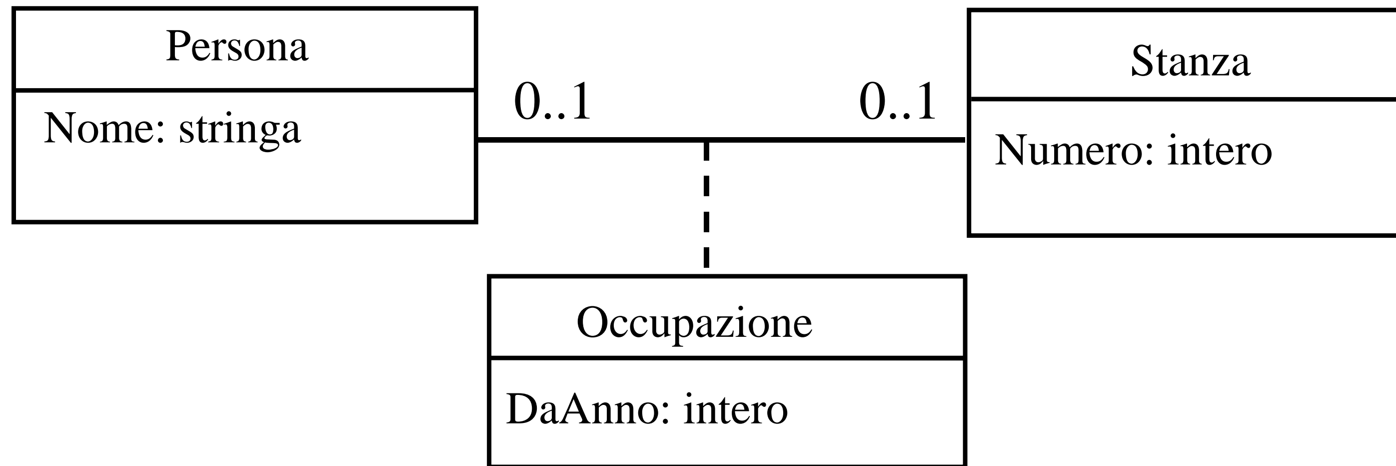
Si possono verificare le funzioni realizzate facendo riferimento al seguente caso di test.

```
-------++----+----+----+----+----+----+----+----+----+----+----+----+-----+
sergio ||    |    |    |    | az1| az1| az1| az1| az1| az1| az2| az2|     |
-------++----+----+----+----+----+----+----+----+----+----+----+----+-----+
aldo   ||    |    | az2| az2| az2|    | az1| az1| az1| az1| az1|    |     |
-------++----+----+----+----+----+----+----+----+----+----+----+----+-----+
mario  || az2| az2| az2| az2| az2|    |    | az1| az1| az1| az1|    |     |
=======++====+====+====+====+====+====+====+====+====+====+====+====+=====+
ANNO   || 89 | 90 | 91 | 92 | 93 | 94 | 95 | 96 | 97 | 98 | 99 | 00 | 01  |
-------++----+----+----+----+----+----+----+----+----+----+----+----+-----+
```

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.)

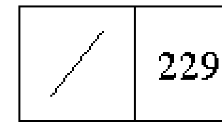
Problema di fondo:
quando creiamo un link fra un oggetto Java `pe` di classe `Persona` un oggetto Java `st` di classe `Stanza`, dobbiamo cambiare lo stato **sia di** `pe` **sia di** `st`.

In particolare:

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

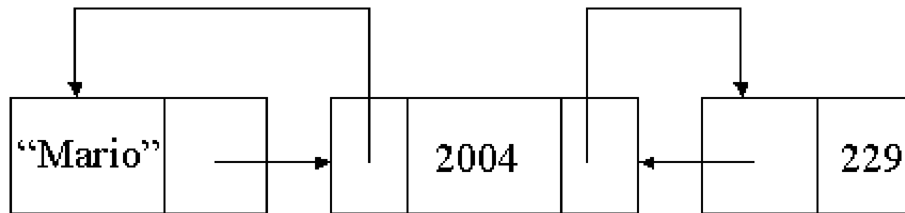
Discorso analogo vale quando **eliminiamo** un link fra due oggetti.

Mantenimento coerenza



Inserimento ⇓

Cancellazione ⇑



Resp. di entrambe le classi UML (cont.)

Ci sono due possibili scelte per garantire la coerenza: (1) demandare questo compito al cliente delle classi `Persona` e `Stanza`; (2) definire meccanismi opportuni per il mantenimento della coerenza nelle classi `Persona` e `Stanza`;

Per ovvi motivi è preferibile seguire l'approccio (2), dato che non possiamo lasciare in linea di principio al cliente un compito così oneroso. Per realizzare questo approccio è necessario **centralizzare** la responsabilità di assegnare i riferimenti in maniera corretta.

Questo può essere ottenuto attraverso la realizzazione di una opportuna classe Java, chiamata ad esempio `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*.

Resp. di entrambe le classi UML (cont.)

Nonostante l'approccio (2) sia preferibile, nel seguito, per motivi di semplicità adotteremo l'approccio (1). In altri termini, nei casi in cui la responsabilità su un'associazione sia doppia, lasceremo ai clienti che manipolano istanze delle classi coinvolte nell'associazione il compito di mantenere i riferimenti coerenti.

Questa scelta implica di fatto che per realizzare un'associazione A fra le classi C e D aventi entrambi responsabilità su A, si può procedere considerando singolarmente la responsabilità di C e di D, cioè come se C e D avessero entrambe responsabilità singola su A. Sarà invece il cliente ad operare in maniera diversa dal caso di responsabilità singola, avendo cura di mantenere coerenti i riferimenti. Nel nostro esempio, quindi, nel realizzare le classi *Stanza* e *Persona* procederemo come fatto nel caso di responsabilità singola, e molteplicità 0..1.

Come già visto nel caso di responsabilità singole, utilizzeremo nel seguito una classe Java per i link nel caso siano presenti attributi di associazione. Nel nostro esempio, realizzeremo la classe `TipoLinkOccupazione`, che modella tuple del prodotto cartesiano tra *Stanza* e *Persona* con attributo *DaAnno*.

Classe Java Persona

```
// File RespEntrambi01New/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 (link == null && t != null && t.getPersona()==this)
        link = t;
  }
  public void eliminaLinkOccupazione() {
       link = null;
  }
  public TipoLinkOccupazione getLinkOccupazione() {
      return link;
  }
}
```

Classe Java Stanza

```
// File RespEntrambi01New/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 (link == null && t != null && t.getStanza()==this)
        link = t;
  }
  public void eliminaLinkOccupazione() {
        link = null;
  }
  public TipoLinkOccupazione getLinkOccupazione() {
      return link;
  }
}
```

Classe Java TipoLinkOccupazione

```
// File RespEntrambi01New/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; }
}
```

Inserimento e Cancellazione 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".

Come fatto per il caso di responsabilità singola e molteplicità 0..1, nelle funzioni di tipo inserisciLink() viene prima controllato che il campo link sia nullo. Nel caso il link sia già presente, bisognerà esplicitamente cancellarlo per poi inserirne uno nuovo.

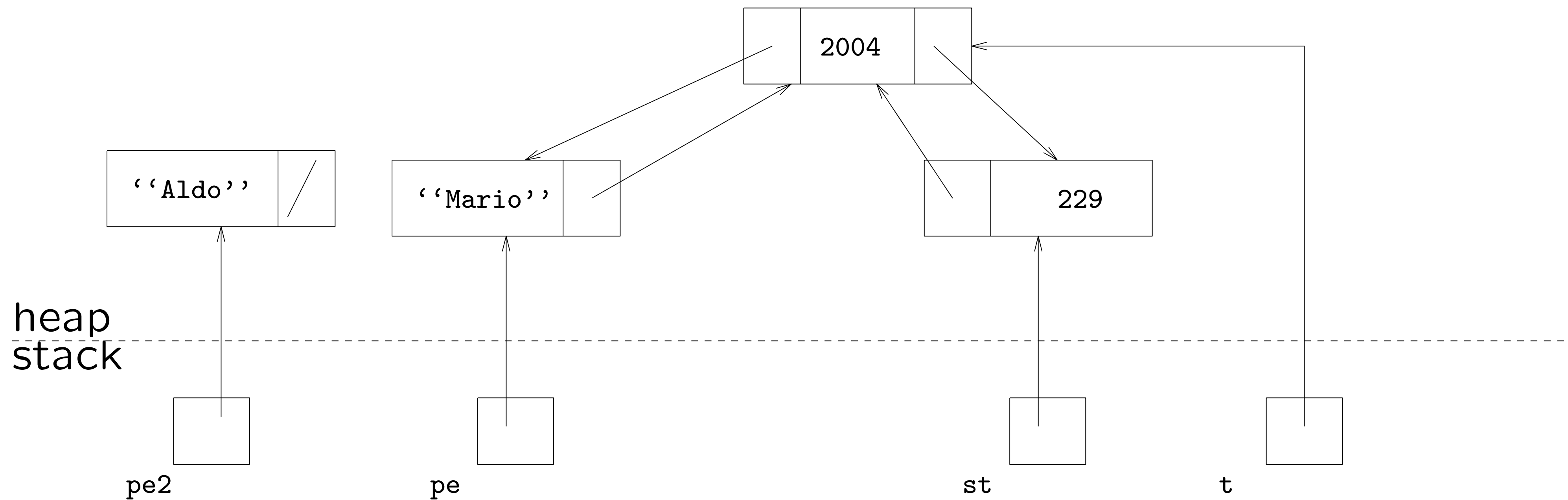
Il client, dal momento che deve essere sicuro che l'inserimento di un link sia avvenuto in entrambi gli oggetti coinvolti nel link, farà un ulteriore controllo prima di effettuare inserimenti sfruttando la funzione getLink.

Inserimento e Cancellazione di link: controlli

Nella cancellazione di un link fra due oggetti o_1 ed o_2, il cliente si preoccuperà di modificare opportunamente o_1 ed o_2, in modo che dopo la cancellazione entrambi non risultino più coinvolti nel link cancellato.

Nell'inserimento di un link fra due oggetti o_1 ed o_2 che non sono coinvolti in un altro link, il cliente si preoccuperà di modificare opportunamente tutti e due gli oggetti, in modo che dopo l'inserimento, risultino entrambi coinvolti nel link.

Possibile stato della memoria



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

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);
  }
pe.inserisciLinkOccupazione(t);
st.inserisciLinkOccupazione(t);
```

Aggiornamento del link

Supponiamo adesso di assegnare la stanza 229 ad Aldo dal 2004, e di lasciare Mario senza stanza assegnata

```
TipoLinkOccupazione t2 = null;
try { t2 = new TipoLinkOccupazione(st,pe2,2004); }
catch (EccezionePrecondizioni e) { System.out.println(e);  }
//se Aldo ha gia' una stanza assegnata si elimina il relativo link
// sia da Aldo che dalla stanza coinvolta in questo link
//(nel nostro caso il link non c'e' ma in genere può essere presente)
if(pe2.getLinkOccupazione() != null) {
  pe2.getLinkOccupazione().getStanza().eliminaLinkOccupazione();
  pe2.eliminaLinkOccupazione();  }

//si elimina il link fra Mario e la stanza 229
p.eliminaLinkOccupazione();
st.eliminaLinkOccupazione();

//si inserisce infine il nuovo link fra Aldo e la stanza 229
pe2.inserisciLinkOccupazione(t2);
st.inserisciLinkOccupazione(t2);
```

Esercizio 8: 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; la stanza *st* non è assegnata

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

FineSpecifica

Soluzione esercizio 8

```
// 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) {
                p.getLinkOccupazione().getStanza().eliminaLinkOccupazione();
                p.eliminaLinkOccupazione();
                TipoLinkOccupazione t = null;
```

```
            try {
                t = new TipoLinkOccupazione(st,p,anno);
            }
            catch (EccezionePrecondizioni e) {
                System.out.println(e);
            }
            p.inserisciLinkOccupazione(t);
            st.inserisciLinkOccupazione(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) {
```

```
                s.getLinkOccupazione().getPersona().eliminaLinkOccupazione();
                s.eliminaLinkOccupazione();
            }
        }
    }

private RiallocazionePersonale() { }
}
```

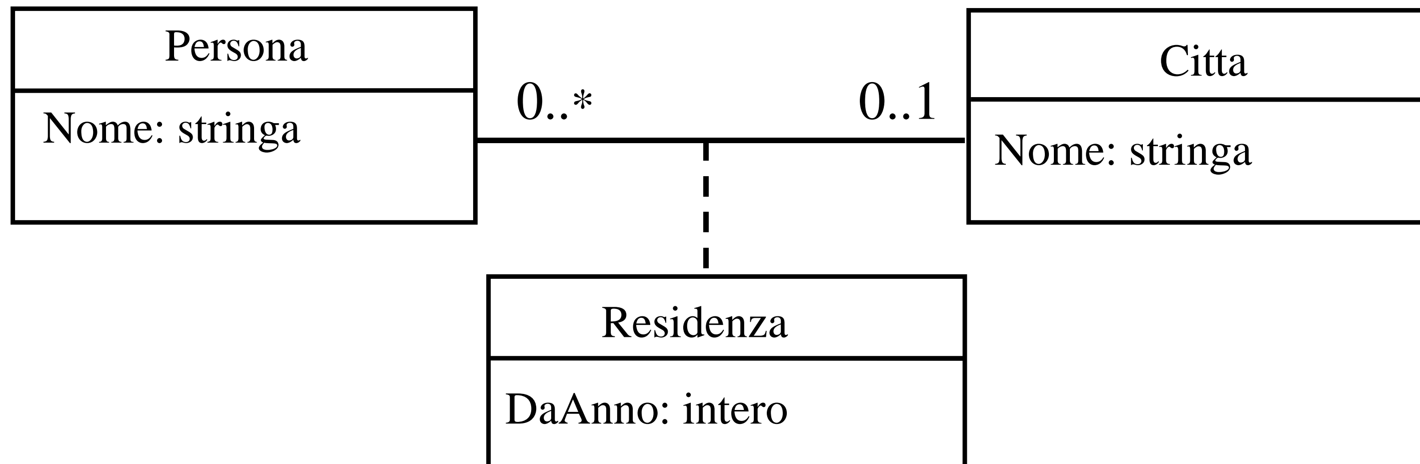
Resp. di entrambe le classi UML: molt. 0..*

Le considerazioni fatte per il caso di associazione binaria con responsabilità doppia e con entrambe le molteplicità pari a 0..1 valgono anche per il caso di molteplicità pari a 0..* (per un solo ruolo o per entrambi). In altri termini, anche per questi casi le due responsabilità vengono realizzate in maniera separata, procedendo analogamente a quanto fatto nel caso di responsabilità singola.

Anche in questo caso saranno i clienti delle classi ad assicurarsi che i riferimenti agli oggetti siano trattati in maniera coerente. Come nel caso di molteplicità 0..1 visto in precedenza, una scelta più rigorosa (ed efficace) è quella di gestire con opportuni meccanismi questa coerenza lato server, ma, per semplicità, in queste slide viene proposto solo l'approccio lato client.

Per completezza, nel seguito vediamo due esempi con associazioni binarie, responsabilità doppia, e molteplicità 0..* (almeno in un ruolo).

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



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..*

Come nel caso di responsabilità singola, 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 dati 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;

Classe Java Citta

```
// File RespEntrambi0STARClient/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)
         insieme_ink.add(t);
    }
    public void eliminaLinkResidenza(TipoLinkResidenza t) {
      if (t != null && t.getCitta()==this)
         insieme_ink.remove(t);
    }
    public Set<TipoLinkResidenza> getLinkResidenza() {
        return (HashSet<TipoLinkResidenza>)insieme_link.clone();
    }
}
```

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 (ed analoga al corrispondente caso con responsabilità singola).

- 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 RespEntrambi0STARClient/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 (link == null && t != null && t.getPersona()==this)
         link = t;
  }
    public void eliminaLinkResidenza() {
         link = null;
  }
  public TipoLinkResidenza getLinkResidenza() {
    return link;
  }
}
```

Classe Java TipoLinkResidenza

```
// File RespEntrambi0STARClient/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; }
}
```

Esempio

Il seguente codice si riferisce ad una situazione in cui Mario ed Aldo sono entrambi residenti a Roma

```
  Citta c = new Citta("Roma");

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

  TipoLinkResidenza t = null;
  TipoLinkResidenza t2 = null;
  try {
      t = new TipoLinkResidenza(c,pe,2006);
      t2 = new TipoLinkResidenza(c,pe2,2007);
  }
  catch (EccezionePrecondizioni e) {
      System.out.println(e);
  }
c.inserisciLinkResidenza(t);
pe.inserisciLinkResidenza(t);
c.inserisciLinkResidenza(t2);
pe2.inserisciLinkResidenza(t2);
```

Esempio

Il seguente codice descrive la situazione in cui Aldo cambia residenza e va a Milano

```
  Citta c2 = new Citta("Milano");

  TipoLinkResidenza t3 = null;
  try {
      t3 = new TipoLinkResidenza(c2,pe2,2007);
  }
  catch (EccezionePrecondizioni e) {
      System.out.println(e);
  }
//si rimuove prima la vecchia residenza
pe2.getLinkResidenza().getCitta().rimuoviLinkResidenza(t2);
pe2.eliminaLinkResidenza();

c2.inserisciLinkResidenza(t3);
pe2.inserisciLinkResidenza(t3);
```

Esercizio 9: 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

Soluzione esercizio 9 (1/2)

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;
```

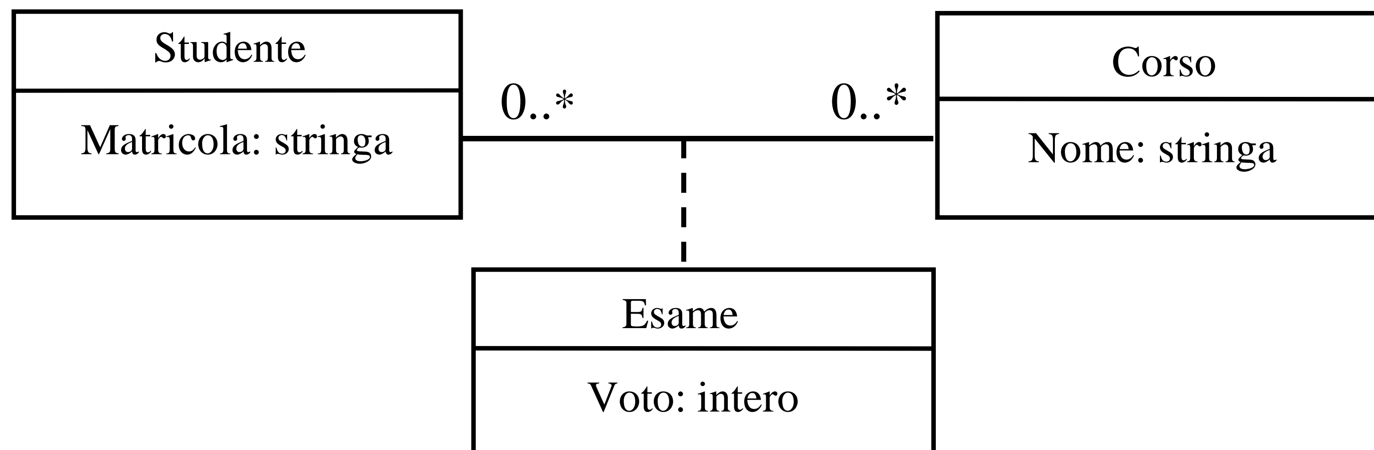
Soluzione esercizio 9 (2/2)

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

```
// File RespEntrambi0STAR/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() { }
}
```

Entrambe le molteplicità sono 0..*

Caso di associazioni binarie in cui **entrambe le classi hanno responsabilità sull'associazione, ed entrambe con molteplicità 0..***.



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

In questo caso, le due classi Java che realizziamo sono strutturalmente simili, avendo entrambe molteplicità 0..* nell'associazione su cui hanno responsabilità.

Classe Java Studente

```
// File RespEntrambi0STAR2Client/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)
            insieme_link.add(t);
    }
    public void eliminaLinkEsame(TipoLinkEsame t) {
         if (t != null && t.getStudente()==this)
            insieme_link.remove(t);
    }
     public Set<TipoLinkEsame> getLinkEsame() {
        return (HashSet<TipoLinkEsame>)insieme_link.clone();
    }
}
```

Classe Java Corso

```
// File RespEntrambi0STAR2Client/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)
            insieme_link.add(t);
    }
    public void eliminaLinkEsame(TipoLinkEsame t) {
        if (t != null && t.getCorso()==this)
            insieme_link.remove(t);
    }
    public Set<TipoLinkEsame> getLinkEsame() {
        return (HashSet<TipoLinkEsame>)insieme_link.clone();
    }
}
```

Classe Java TipoLinkEsame

```
// File RespEntrambi0STAR2Client/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; }
}
```

Esercizio 10: 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

Soluzione esercizio 10 (1/2)

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.

Soluzione esercizio 10 (2/2)

```
// File RespEntrambi0STAR2/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() { }
}
```

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.

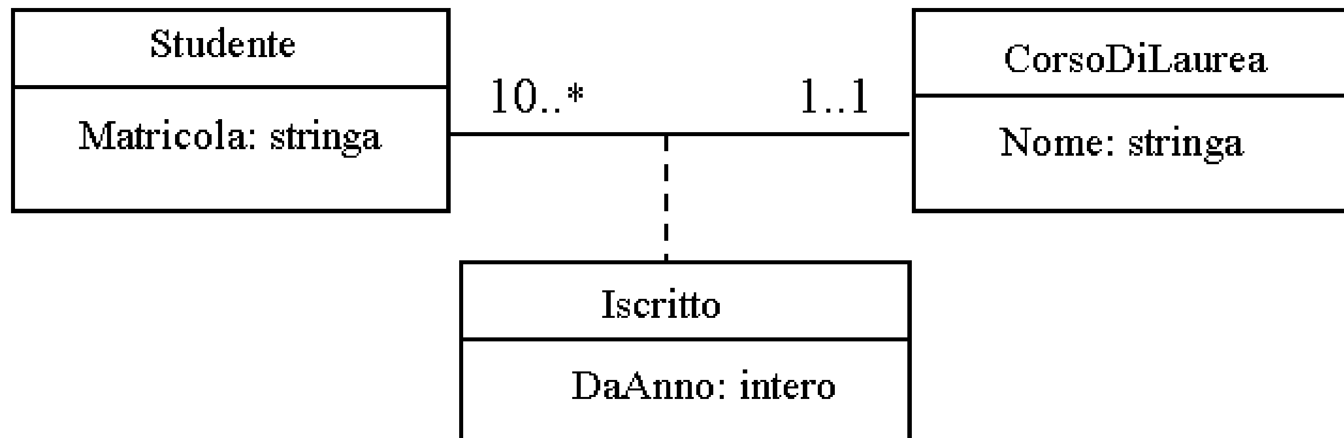
Come già chiarito nella fase di progetto, in generale prevediamo che la classe Java rispetto a cui esiste uno dei vincoli di cui sopra **abbia necessariamente responsabilità sull'associazione**. Il motivo, che verrà chiarito in seguito, è che gli oggetti di tale classe classe **devono poter essere interrogati**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

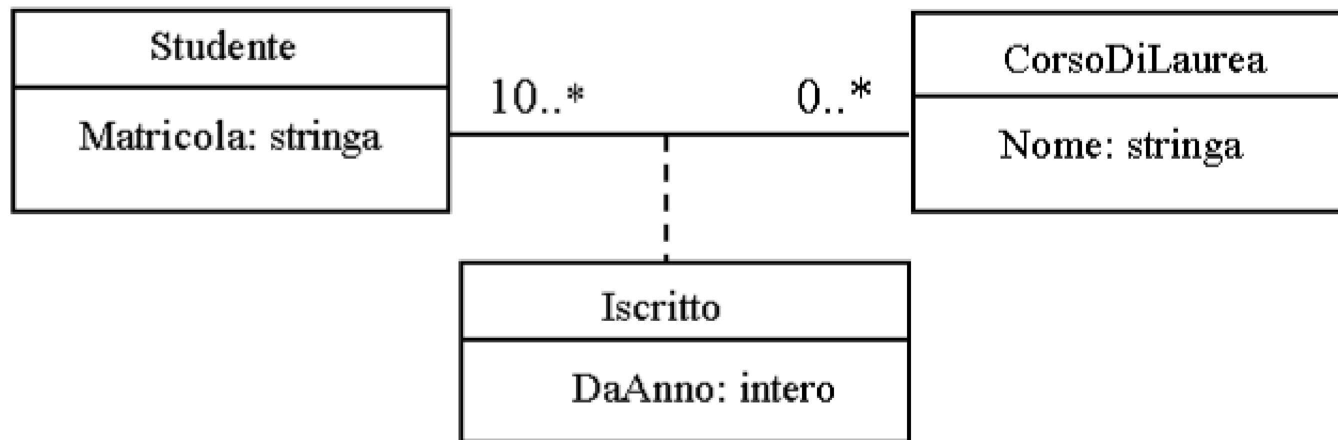
Infatti, uno studente potrebbe nascere solamente nel momento in cui esiste già un corso di laurea, ma un corso di laurea deve avere almeno dieci studenti, e questo indica una intrinseca complessità nel creare oggetti, e al tempo stesso fare in modo che essi non violino vincoli di cardinalità minima.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

Molteplicità minima diversa da zero (cont.)

- Rispetto al caso di associazione con responsabilità singola 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 singola e vincoli di molteplicità entrambi 0..*, la metodologia di realizzazione delle seguenti classi Java:

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

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)   insieme_link.add(t);     }
    public void eliminaLinkIscritto(TipoLinkIscritto t) {
      if (t != null && t.getCorsoDiLaurea()==this)   insieme_link.remove(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();     }
}
```

Esercizio 11

Realizzare in Java il cliente *Valutazione Didattica*(simile al cliente visto nell'esercizio 12):

InizioSpecificaOperazioni Valutazione Didattica

CorsoFacile *(c: Corso): booleano*

pre: nessuna

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

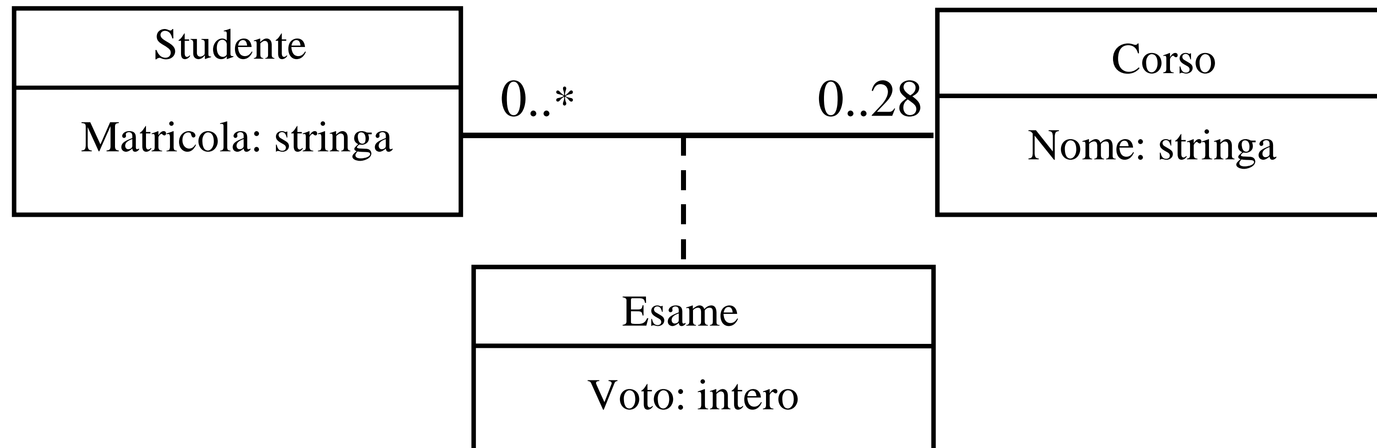
FineSpecifica

Soluzione esercizio 11

L'algoritmo ricalca quello fornito come soluzione per l'esercizio 10.

```
// File MoltMax/ValutazioneDidattica.java

import java.util.*;

public final class ValutazioneDidattica {
    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() { }
}
```

Molteplicità massima finita



Supponiamo che solo *Studente* abbia responsabilità sull'associazione.

Molteplicità massima finita (cont.)

- Rispetto al caso di associazione con responsabilità singola 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 singola e vincoli di molteplicità entrambi 0..* le seguenti classi Java:

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

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() { return insieme_link.size(); // FUNZIONE NUOVA  }
    public void inserisciLinkEsame(tipoLinkEsame t)  {
        if (t != null && t.getEsame()==this)
           insieme_link.add(t);     }
    public void eliminaLinkEsame(tipoLinkEsame t) {
        if (t != null && t.getEsame()==this)
           insieme_link.remove(t);     }
    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 12

Realizzare in Java il cliente *Valutazione Didattica*(simile al cliente visto nell'esercizio 11):

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

FineSpecifica

Soluzione esercizio 12

```
// File MoltMax/ValutazioneDidatticaBis.java

import java.util.*;

public final class ValutazioneDidatticaBis {
    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;
    }

    private ValutazioneDidatticaBis() { }
}
```

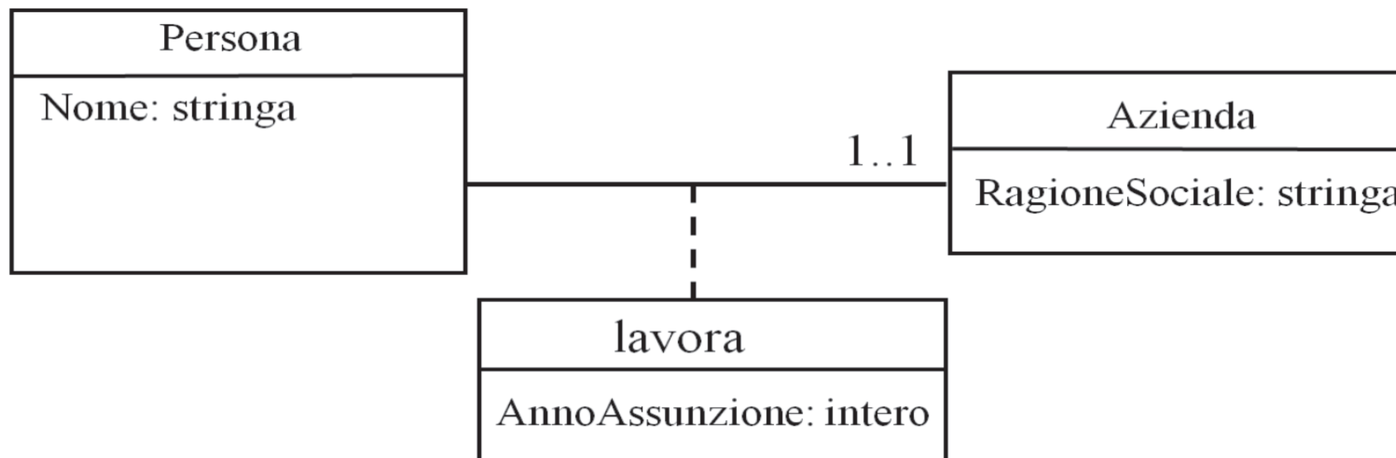
Molteplicità massima 1

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.)

Riprendiamo un esempio già visto, e assumiamo che in questo caso vi sia una molteplicità 1..1:



Assumiamo che solo *Persona* abbia responsabilità sull'associazione.

Molteplicità massima 1 (cont.)

- Rispetto al caso di associazione con responsabilità singola e in cui il vincolo di molteplicità della classe che ha responsabilità sull'associazione sia 0..1, la classe Java `Persona` si differenzia nei seguenti aspetti:

  1. Ha un'ulteriore funzione pubblica `int quantiLavora()`, che restituisce il numero di aziende per cui lavora la persona oggetto di invocazione.

  2. La funzione `int getLinkLavora()` lancia una opportuna eccezione (di tipo `EccezioneCardMin`) quando l'oggetto di invocazione non rispetta il vincolo di cardinalità minima sull'associazione *Lavora*.

Molteplicità massima 1 (cont.)

Rimangono invece inalterate, rispetto al caso di associazione con responsabilità singola e vincoli di molteplicità 0..1 e 0..* le seguenti classi Java:

- `Azienda`,
- `TipoLinkLavora`,
- `EccezionePrecondizioni`.

Il codice della classe `EccezioneCardMin` è già stato dato in pecedenza, per cui nel seguito riportiamo solo il codice della classe `Persona`.

Classe Java Persona

```
// File MoltMax1/Persona.java
public class Persona {
    private final String nome;
    private TipoLinkLavora link;
    public static final int MIN_LINK_LAVORA = 1;
    public Persona(String n) { nome = n; }
    public String getNome() { return nome; }
    public int quantiLavora() { // FUNZIONE NUOVA
        if (link == null) return 0;
        else return 1;     }
    public void inserisciLinkLavora(TipoLinkLavora t) {
        if (link == null && t != null && t.getPersona() == this)
            link = t;     }
    public void eliminaLinkLavora() {
        link = null;     }
    public TipoLinkLavora getLinkLavora() throws EccezioneCardMin {
        if (link == null)
           throw new EccezioneCardMin("Cardinalita’ minima violata");
        else return link;     }
}
```

Esercizio 13: cliente

Realizzare in Java il cliente *Cambiamento Azienda*:

InizioSpecificaOperazioni Cambiamento Azienda

Cambiano *(i: Insieme(Persona), a: Azienda, x: intero)*
pre: nessuna
post: ogni persona di i lavora per l'azienda a a partire dall'anno x

FineSpecifica

Soluzione esercizio 13

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

```
per ogni persona p di i
    se p non lavora per l'azienda a
        allora elimina il link di tipo lavora da p;
               crea un nuovo link di tipo lavora per p,
               con l'azienda a e l'anno x;
```

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

Soluzione esercizio 13 (2/2)

```
// File MoltMax1/CambiamentoAzienda.java
import java.util.*;
public final class CambiamentoAzienda {
  public static void cambiano(Set<Persona> i, Azienda a, int h) throws EccezioneCardMin {
        Iterator<Persona> it = i.iterator();
        while(it.hasNext()) {
            Persona p = it.next();
            if (p.getLinkLavora().getAzienda() != a) {
                p.eliminaLinkLavora();
                TipoLinkIscritto t = null;
                try {
                    t = new TipoLinkLavora(a,p,h);
                }
                catch (EccezionePrecondizioni e) {
                    System.out.println(e);
                }
                p.inserisciLinkLavora(t);
            }
        }
  }
  private CambiamentoAzienda() { }
}
```

ERRORE!

Associazioni n-arie

Si trattano generalizzando quanto visto per le associazioni binarie.

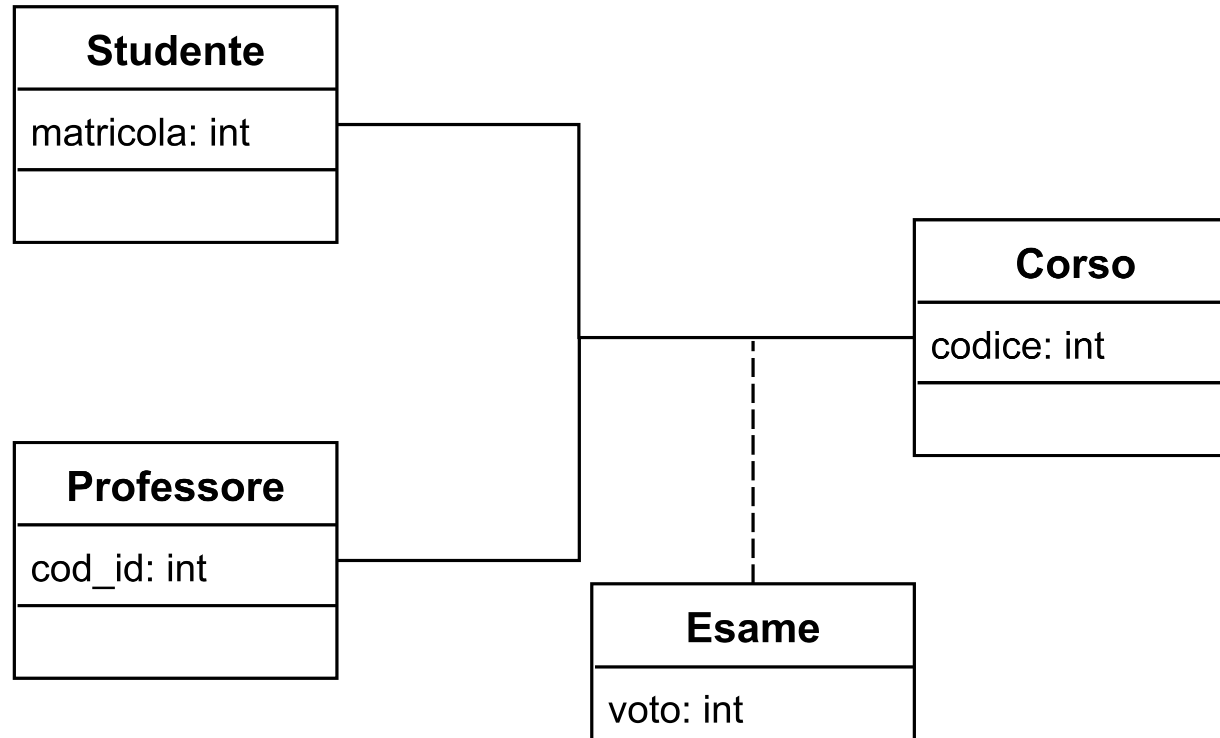
Ricordiamo che noi assumiamo che le molteplicità delle associazioni n-arie siano sempre 0..*.

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 di una classe, si prevede la struttura di dati per rappresentare i link in ciascuna classe che abbia responsabilità. Anche in questo caso adottiamo la semplificazione di lasciare l'onere del controllo della coerenza dei riferimenti ai clienti delle classi realizzate.

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`;
  - `Studente`/`Professore`;
  - `Corso`.

Esempio: la classe Java Studente

```
import java.util.*;
public class Studente {
    private final int matricola;
    private HashSet<TipoLinkEsame> insieme_link;
    public Studente(int n) {
        matricola = n;
        insieme_link = new HashSet<TipoLinkEsame>();
    }
    public int getMatricola() { return matricola; }
    public void inserisciLinkEsame(TipoLinkEsame t) {
        if (t != null && t.getStudente()==this)
            insieme_link.add(t);
    }
    public void eliminaLinkEsame(TipoLinkEsame t) {
         if (t != null && t.getStudente()==this)
            insieme_link.remove(t);
    }
     public Set<TipoLinkEsame> getLinkEsame() {
        return (HashSet<TipoLinkEsame>)insieme_link.clone();
    }
}
```

Esempio: la classe Java Professore

```
import java.util.*;
public class Professore {
    private final int codId;
    private HashSet<TipoLinkEsame> insieme_link;
    public Professore(int n) {
        codId = n;
        insieme_link = new HashSet<TipoLinkEsame>();
    }
    public int getCodId() { return codId; }
    public void inserisciLinkEsame(TipoLinkEsame t) {
        if (t != null && t.getStudente()==this)
            insieme_link.add(t);
    }
    public void eliminaLinkEsame(TipoLinkEsame t) {
         if (t != null && t.getStudente()==this)
            insieme_link.remove(t);
    }
     public Set<TipoLinkEsame> getLinkEsame() {
        return (HashSet<TipoLinkEsame>)insieme_link.clone();
    }
}
```

Esempio: la classe Java Corso

```
public class Corso {
    private final int codice;
    public Corso(String n) {
        codice = n;
    }
    public int getCodice() { return codice; }
}
```

Esempio: la classe Java EccezionePrecondizioni

```
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;
  }
}
```

Esempio: la classe Java TipoLinkEsame

```
public class TipoLinkEsame {
    private final Corso ilCorso;
    private final Studente loStudente;
private final Professore ilProf;
    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; ilProf = 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.ilProf == ilProf; }
        else return false;
        }
    public int hashCode() { return ilCorso.hashCode() + loStudente.hashCode() + ilProf.hashC
    public Corso getCorso() { return ilCorso; }
    public Studente getStudente() { return loStudente; }
public Professore getProfessore() { return ilProf; }
    public int getVoto() { return voto; }
}
```

Associazioni Ordinate

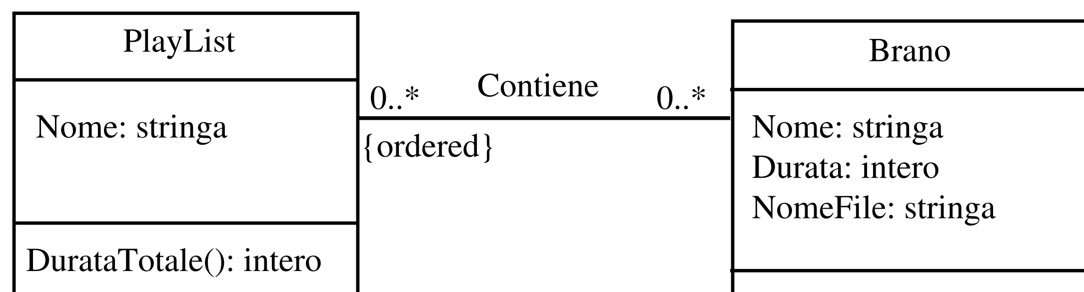
- Le associazioni ordinate si realizzano in modo del tutto analogo alle relazioni non ordinate.

- Per mantenere l'ordinamento però si fa uso delle classi `List` e `LinkedList` invece di `Set` e `HashSet`.

- Si noti 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

```
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 in Java lo use case *Analisi PlayList*, specificato di seguito:

InizioSpecificaUseCase 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

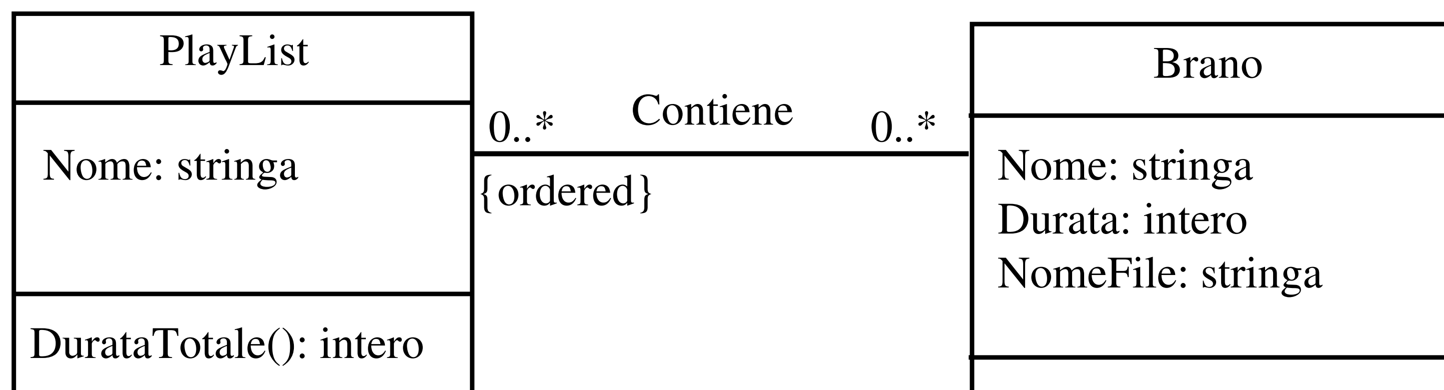
Realizzazione in Java del cliente

La progettazione dell'algoritmo è lasciata come esercizio

```
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 prevedere:
  - 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. **La classe `PlayList` è identica al caso precedente**
  - realizzare i clienti in modo tale da mantenere coerenti i riferimenti.

Realizzazione in Java della classe Brano

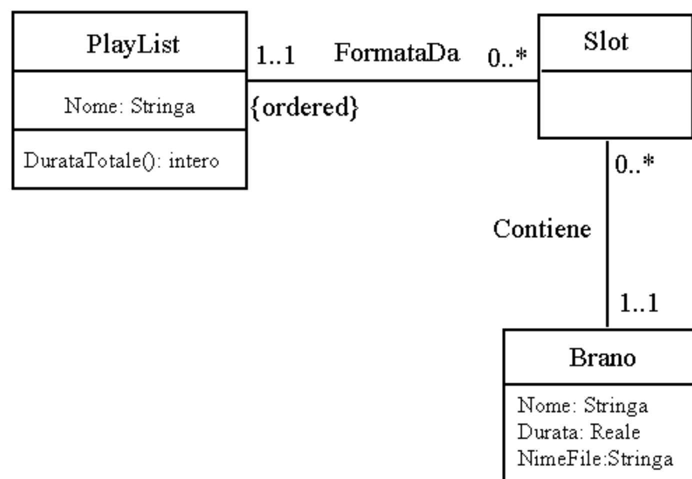
```
// File OrdinateEntrambiOSTARClient/Brano.java
import java.util.*;
public class Brano {
    private final String nome;
    private final int durata;
    private final String nomefile;
    private HashSet<PlayList> insieme_link;
    public Brano(String n, int d, String f) {
        nome = n;
        durata = d;
        nomefile = f;
        insieme_link = new HashSet<PlayList>();
    }
    public String getNome() { return nome; }
    public int getDurata() { return durata; }
```

```
    public String getNomeFile() { return nomefile; }
    public void inserisciLinkContiene(PlayList p) {
      if (p != null) insieme_link.add(p);
    }
    public void eliminaLinkContiene(PlayList p) {
      if (p != null) insieme_link.remove(p);
    }
    public Set<PlayList> getLinkContiene() {
        return (HashSet<PlayList>)insieme_link.clone();
    }
}
```

Esercizio 18

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

- **solo Slot ha responsabilità su Contiene**;
- sia *PlayList* che *Slot* hanno responsabilità su *FormataDa*.



Esercizio 18 (cont.)

Nella realizzazione, possiamo tenere conto che (cfr. *Seconda Parte* del corso) 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*.

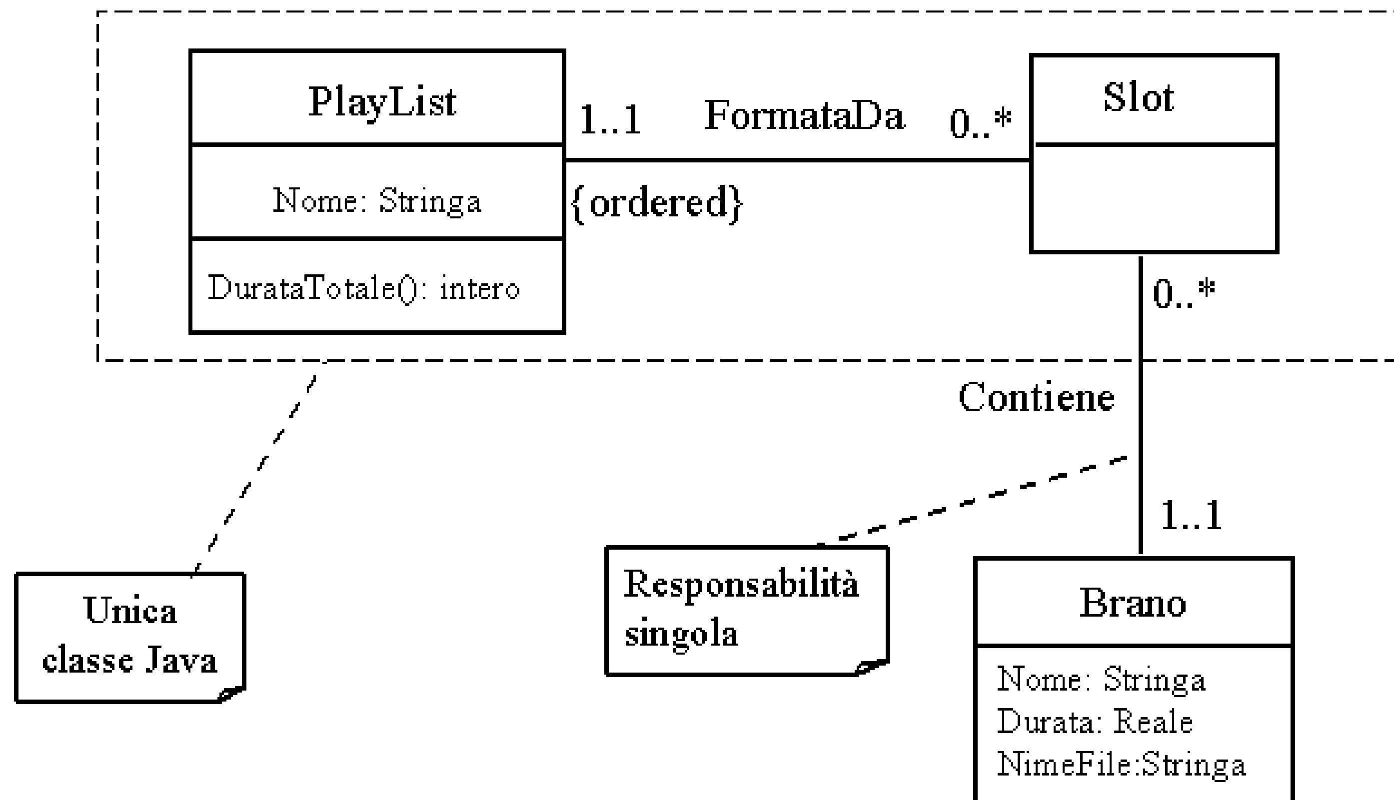
Soluzione esercizio 18

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 use case sono identiche al caso della responsabilità singola.

Soluzione esercizio 18 (cont.)



Sol. eserc. 18: 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);
    //in questo caso non controlliamo se il brano è gia presente
  }
  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;
  }
}
```

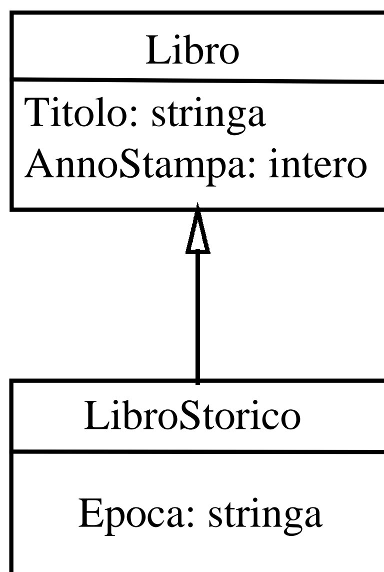
Realizzazione di generalizzazioni

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.



Assumiamo che tutte le proprietà in entrambe le classi siano immutabili (e note dalla nascita)

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

```
                             =============================
                             IL METODO B VEDE IL CAMPO A ?
                             =============================
METODO B \          +--------+-----------+-----------+---------+
 IN       \CAMPO A | public | protected | non qual. | private |
-----------\-------+--------+-----------+-----------+---------+
                   +--------+-----------+-----------+---------+
STESSA CLASSE      |   SI   |    SI     |    SI     |   SI    |  1 |
                   |        |           |           |         |    |
                   +--------+-----------+-----------+---------+    |
CLASSE STESSO      |   SI   |    SI     |    SI     |   NO    |  2 |   NOTA:
PACKAGE            |        |           |           |         |    | Decrescono
                   +--------+-----------+-----------+---------+    | i diritti
CLASSE DERIVATA    |   SI   |    SI     |    NO     |   NO    |  3 |
PACKAGE DIVERSO    |        |           |           |         |    |
                   +--------+-----------+-----------+---------+    |
CL. NON DERIVATA   |   SI   |    NO     |    NO     |   NO    |  4 |
PACKAGE DIVERSO    |        |           |           |         |    V
                   +--------+-----------+-----------+---------+    V

                    ---------------------------------------->>
                                NOTA: Decrescono i diritti
```

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

InizioSpecificaUseCase 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
```

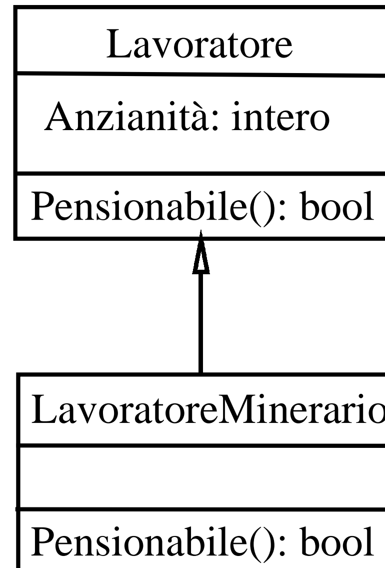
Ridefinizione

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**, ed **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

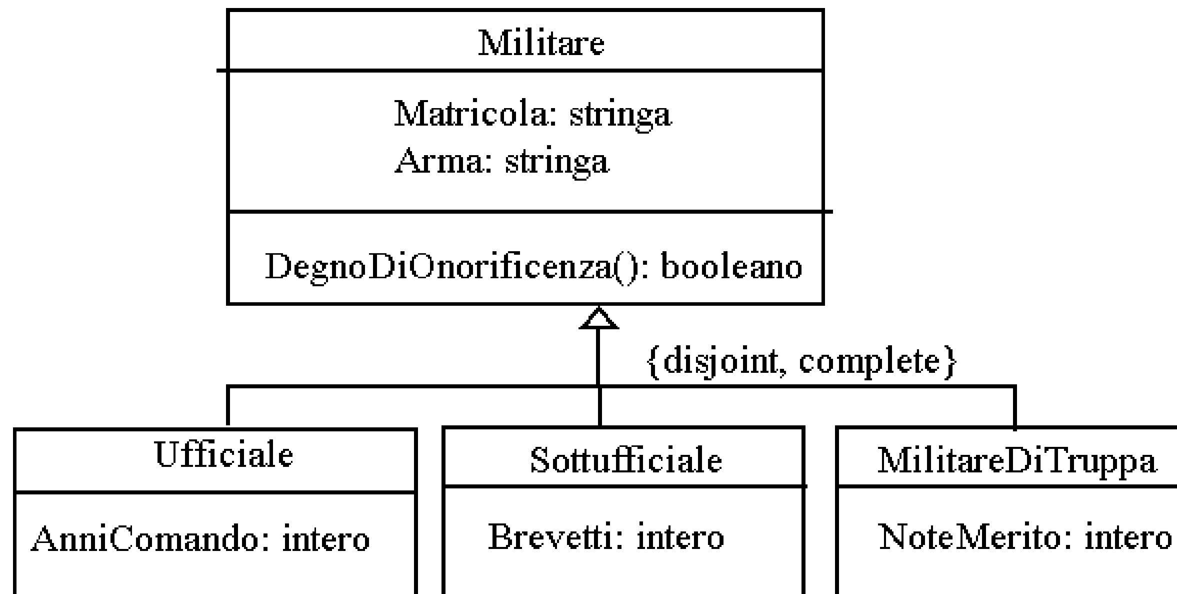
```
// File Generalizzazione/LavoratoreMinerario/LavoratoreMinerario.java

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.)



Nota: gli attributi `AnniComando`,`Brevetti` e `NoteMerito`, sono modificabili solo tramite le funzioni `IncrementaAnniComando`, `IncrementaBrevetti`, `IncrementaNoteMerito`.

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:

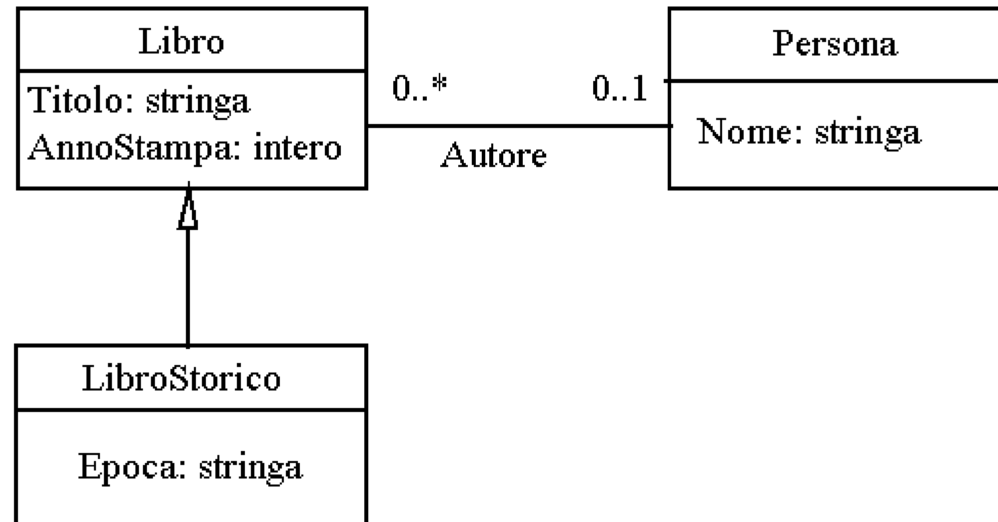
- Tutta l'applicazione viene messa in un package Java `P`, nel direttorio `P`.
- Ogni classe Java dell'applicazione proveniente dal diagramma delle classi (anche quelle definite per le associazioni) viene messa nel package `P`.
- Ciò vale anche per quelle definite per i tipi, a meno che siano in opportuni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

- Ogni classe Java proveniente dal diagramma degli use case viene messa nel package `P`.

Packages, esempio



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

Packages, esempio (cont.)

InizioSpecificaUseCase 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;
  }
}
```

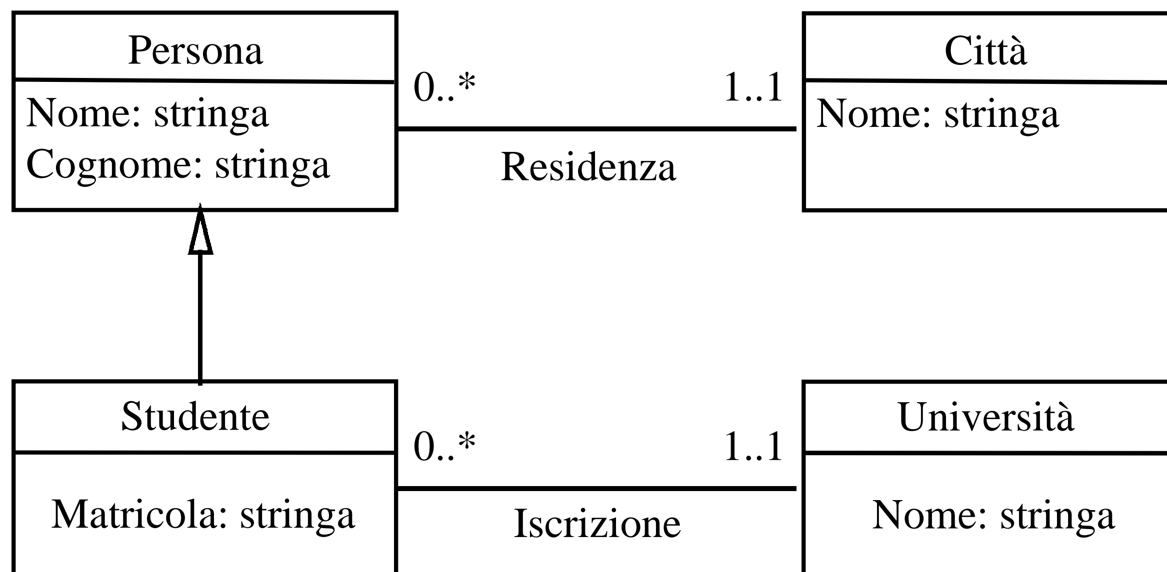
La classe Java LibroStorico

```
// 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 19



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.

Soluzione esercizio 19

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
|   |   EccezioneCardMin.java
|   |
|   +---Persona
|   |       Persona.java
|   |
|   \---Studente
|           Studente.java
```

Sol. eserc. 19: classe Java Universita

```
package PackageUniversita;
import java.util.*;
import PackageUniversita.Studente.Studente;

public class Universita {
 private final String nome;
 private HashSet<Studente> insieme_link;
 public Universita(String n) {
     nome = n;
     insieme_link = new HashSet<Studente>();
 }
 public String getNome() { return nome; }
 public void inserisciLinkIscritto(Studente t) {
    if (t != null) insieme_link.add(t);
 }
 public void eliminaLinkIscritto(Studente t) {
    if (t != null) insieme_link.remove(t);
 }
 public Set<Studente> getLinkIscrizione() {
    return (HashSet<Studente>)insieme_link.clone();
 }
}
```

Sol. eserc. 19: classe Java Citta

```
package PackageUniversita;

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

Sol. eserc. 19: classe Java Persona

```
package PackageUniversita.Persona;
import PackageUniversita.Citta;
import PackageUniversita.EccezioneCardMin;
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. 19: classe Java Studente

```
package PackageUniversita.Studente;
import PackageUniversita.*;
import PackageUniversita.Persona.*;
public class Studente extends Persona {
    protected final String matricola;
    protected Universita iscrizione;
    public static final int MIN_LINK_ISCRIZIONE = 1;
    public Studente(String n, String c, String m) {
        super(n,c);
        matricola = m;     }
    public String getMatricola() { return matricola; }
    public int quantiIscrizione() {
        if (iscrizione == null) return 0;
        else return 1;     }
    public void inserisciLinkIscrizione(Universita t) {
       if (t != null ) iscrizione = t;     }
    public void eliminaLinkIscrizione() {
        iscrizione = null;     }
    public Universita getLinkIscrizione() throws EccezioneCardMin {
        if (iscrizione == null)
            throw new EccezioneCardMin("Cardinalita' minima violata");
       else return iscrizione;     }
}
```

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.