

Corso di
PROGETTAZIONE DEL SOFTWARE I
Laurea in Ingegneria Informatica
Prof. Giuseppe De Giacomo
A.A. 2007/08

QUARTA PARTE:

LA FASE DI REALIZZAZIONE

(versione preliminare)

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 e del diagramma degli stati e delle transizioni.

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**.

2

Traduzione in Java del diagramma degli use case

Per semplicità, 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.

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

Vedremo in seguito molti esempi di realizzazione di use case.

3

4

Traduzione in Java del diagramma delle classi e del diagramma degli stati e delle transizioni

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

1. **realizzazione di singole classi e dei diagrammi degli stati e delle transizioni,**
2. **realizzazione delle associazioni,**
3. **realizzazione delle generalizzazioni.**

5

Realizzazione di singole classi e del diagramma degli stati e delle transizioni

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

- classe UML a cui non è associato un diagramma degli stati e delle transizioni;
 - classe UML con soli attributi;
 - classe UML con attributi e operazioni;
- classe UML a cui è associato un diagramma degli stati e delle transizioni.

6

Realizzazione di una classe UML con soli attributi

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

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

7

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

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

8

Metodologia per la realizzazione

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

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

9

Metodologia per la realizzazione: campi dati

I campi dati della classe Java *C* corrispondono agli attributi della classe UML *C*.

Le regole principali sono le seguenti:

- I campi dati di *C* sono tutti `private` o `protected`, per incrementare l'information hiding.
- Tali campi sono `final`, se non vengono più cambiati dopo la creazione dell'oggetto (secondo la *tabella delle proprietà immutabili* prodotta nella fase di progetto); altrimenti, non sono `final`.

10

Metodologia per la realizzazione: 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`).

11

Metodologia: funzioni speciali

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

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

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

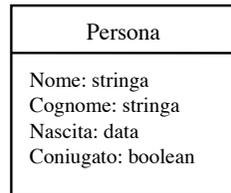
Questa scelta deve essere fatta solo nel caso in cui si vuole che i moduli clienti utilizzino ogni oggetto della classe singolarmente e direttamente – *maggiori dettagli in seguito*.

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

12

Singola classe UML con soli attributi: esempio

Risultato fase di analisi:



Risultato fase di progetto:

Tipo UML	Rappresentazione in Java	Classe UML	Proprietà immutabile
stringa	String	<i>Persona</i>	<i>nome</i>
data	int,int,int		<i>cognome</i>
booleano	boolean		<i>nascita</i>

Proprietà	
Classe UML	nota alla nascita non nota alla nascita

Per default, una persona non è coniugata.

```

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;
}
  
```

13

Realizzazione in Java

// File SoloAttributi/Persona.java

```

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

14

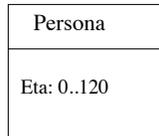
```

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

Il problema dei valori non ammessi

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

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



In tali casi la fase di progetto ha stabilito se dobbiamo utilizzare nella realizzazione un approccio di verifica lato client o lato server.

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

15

Verifica nel lato server: esempio

```
// File SoloAttributi/VerificaLatoServer/Persona.java

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

16

Esempio di cliente

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

InizioSpecificaUseCase **Analisi Statistica**

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

pre: nessuna

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

FineSpecifica

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

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

17

Realizzazione dello use case

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

18

Molteplicità di attributi

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

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

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

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

19

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) {
```

21

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.



20

```
        numTel.remove(n);
    }
    public Set<String> getNumTel() {
        return (HashSet<String>)numTel.clone();
    }
    public String toString() {
        return nome + ' ' + numTel;
    }
}
```

Classe Java Persona: considerazioni

- La classe ha un campo dato di tipo `HashSet`.
- Il costruttore della classe `Persona` crea un oggetto di tale classe, usando 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(String)`: permette di eliminare un numero telefonico;
 - `getNumTel()`: permette di ottenere tutti i numeri telefonici di una persona.

22

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 dalla specifica struttura dati utilizzata per realizzare le funzionalità previste da `Set`, aumentando così l'information hiding.

23

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*.

24

Cliente della classe Java Persona

Per comprendere meglio questo aspetto, consideriamo un cliente della classe `Persona` specificato tramite il seguente use case.

InizioSpecificaUseCase 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

25

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
```

26

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*.

28

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.

27

Esercizio 1: altro cliente della classe

Realizzare in Java il seguente use case *Analisi Recapiti*, specificando prima un algoritmo:

InizioSpecificaUseCase 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

29

Altra realizzazione della classe Java Persona

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

30

Schemi realizzativi

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

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

31

Schema realizzativo con condivisione

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

32

```
}
public Iterator<String> getNumTel() {
    return new IteratoreSolaLettura<String>(numTel.iterator());
}
public String toString() {
    return nome + ' ' + numTel;
}
}
```

La classe Java IteratoreSolaLettura

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

33

Esercizio 2: clienti per la nuova versione della classe Persona

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

34

Realizzazione di classe 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.

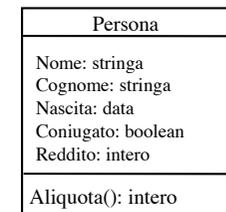
Sono possibili eventuali funzioni `private` o `protected` che dovessero servire per la realizzazione dei metodi della classe *C*, ma che non vogliamo rendere disponibili ai clienti.

35

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.



36

Specifica della classe UML

InizioSpecificaClasse Persona

Aliquota (): intero

pre: nessuna

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

FineSpecifica

```
}
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;
}
```

37

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 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 + "/" +
```

38

```

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

```

Esercizio 3: classi UML con operazioni

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

InizioSpecificaClasse Persona

Aliquota (*data*): *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

39

Esercizio 4: cliente della classe

Realizzare in Java il seguente use case *Analisi Redditi*, specificando prima un algoritmo per la sua operazione:

InizioSpecificaUseCase 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

40

Diagramma degli stati e transizioni

Quando affrontiamo la traduzione in Java di un diagramma degli stati e delle transizioni associato ad una classe UML dobbiamo prendere in considerazione ulteriori aspetti.

- In generale, continuano a valere le regole per la realizzazione di classi enunciate in precedenza.
- Per la rappresentazione degli stati dobbiamo tenere conto della *tabella di rappresentazione degli stati*, prodotta nella fase di progetto.
- Per la realizzazione delle transizioni dobbiamo assicurarci che ogni evento, condizione ed azione corrisponda ad una funzione della classe Java.

41

Campi dati

I campi dati di una classe Java corrispondente ad una classe UML a cui è associato un diagramma degli stati e delle transizioni sono determinati nel seguente modo:

- Continuano a valere le regole per gli attributi viste in precedenza.
- Per la rappresentazione dei valori della tabella di rappresentazione degli stati servono opportune *costanti*, ovvero campi `private`, `static`, `final`.
- Per la rappresentazione dello stato corrente servono **ulteriori** campi dati, del tipo opportuno, come specificato dalla tabella. Tali campi dati saranno `private` o `protected`, e non saranno `final`.
- Poiché lo stato iniziale (del diagramma degli stati e delle transizioni) determina il valore iniziale per questi campi dati, è opportuno che tale valore venga fissato:
 - affidandosi al valore di default di Java, oppure
 - fissandone il valore nella dichiarazione.

42

Eventi

Ogni evento *ev* corrisponde ad una funzione `public ev()` della classe Java.

Si noti che un evento può accadere in vari stati, quindi la funzione Java `ev()` si deve occupare di:

- verificare se la transizione può avvenire, ovvero se:
 - lo stato corrente (codificato mediante le apposite variabili scelte in fase di progetto) è compatibile con l'evento, e
 - è soddisfatta la condizione (vedi in seguito);
- attuare la transizione, ovvero:
 - eseguire l'azione (vedi in seguito), e
 - modificare le variabili di stato in maniera opportuna.

44

Campi funzione

I campi funzione della classe Java sono determinati nel seguente modo:

- Continuano a valere le regole viste in precedenza.
- I campi dati che rappresentano gli stati del diagramma degli stati e delle transizioni non sono gestiti tramite le funzioni **get** e **set**, ma tramite opportune funzioni che corrispondono a eventi, condizioni e azioni.
- Ogni evento, condizione ed azione deve corrispondere ad una funzione della classe Java.
- Esistono funzioni di ispezione dello stato solo se previste in fase di analisi (in quanto necessarie per qualche cliente).

43

Condizioni e azioni

Ogni condizione ed ogni azione del diagramma corrisponde ad una distinta funzione `private` della classe Java.

Infatti dal punto di vista concettuale le condizioni e le azioni non rappresentano servizi pubblici, invocabili da clienti.

Ricordiamo che tutti i servizi erogati da una classe UML devono essere stabiliti in fase di analisi e devono corrispondere a *operazioni* della classe stessa.

Inoltre è opportuno che anche per tali funzioni (al pari di quelle che realizzano operazioni di classe o di use case) venga specificato un algoritmo.

45

Esempio: Libro

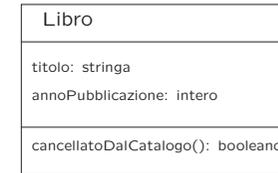
Prenderemo in considerazione il diagramma degli stati e delle transizioni relativo alla classe *Libro*, visto nella seconda parte del corso.



46

Libro: fase di analisi

Supponiamo che sia stata evidenziata una operazione della classe *Libro* che prevede di conoscere se il libro sia stato cancellato dal catalogo oppure no.



InizioSpecificaClasse Libro

cancellatoDalCatalogo() : *booleano*

pre: nessuna

post: *result* vale *true* se *this* è stato cancellato dal catalogo, vale *false*, altrimenti

FineSpecifica

47

Libro: fase di progetto

Assumiamo che in fase di progetto sia stata scelta la seguente rappresentazione per lo stato.

Stato	Rappresentazione in Java	
	tipo var.	int
	nome var.	stato
acquisito	valore	1
in prestito	valore	2
in ritardo	valore	3
finale	valore	4

Assumiamo inoltre il seguente algoritmo per l'azione *invio lettera*

produci una lettera di sollecito che riporti il titolo del libro

48

Libro: realizzazione in Java

// File StatiTransizioni/Libro.java

```

public class Libro {
    private final static int acquisito = 1;
    private final static int in_prestito = 2;
    private final static int in_ritardo = 3;
    private final static int finale = 4;
    private int corrente = acquisito;
    private final String titolo;
    private final int anno_pubblicazione;
    public Libro(String t, int a) {
        titolo = t;
        anno_pubblicazione = a;
    }
    public String getTitolo() { return titolo; }
    public int getAnnoPubblicazione() { return anno_pubblicazione; }
}
  
```

49

```

public void prestito() {
    if (corrente == acquisito)
        corrente = in_prestito;
}
public void scadenzaTermini() {
    if (corrente == in_prestito)
        corrente = in_ritardo;
}
public void restituzione() {
    if (corrente == in_prestito || corrente == in_ritardo)
        corrente = acquisito;
}
public void sollecito() {
    if (corrente == in_ritardo)
        invioLettera();
}
public void cancellazioneCatalogo() {
    corrente = finale;
}

```

Esercizio 5: cliente della classe

Realizzare in Java il seguente use case *Valutazione Biblioteca*, specificando prima un algoritmo per le sue operazioni:

InizioSpecificaUseCase Valutazione Biblioteca

PiùVecchioInCatalogo (*i: Insieme(Libro)*): *Libro*

pre: *i* contiene almeno un libro in catalogo

post: *result* è il libro più vecchio tra quelli di *i* in catalogo

EtàMediaInCatalogo (*i: Insieme(Libro)*, *a: intero*): *Libro*

pre: *i* contiene almeno un libro in catalogo

post: *result* è l'età media all'anno *a* (in anni) dei libri di *i* in catalogo

FineSpecifica

```

}
public boolean cancellatoDalCatalogo() {
    return corrente == finale;
}
private void invioLettera() {
    System.out.println
        ("Gentile Utente, e' invitato a restituire il libro " + titolo);
}
}
}

```

Esercizio 6: diagramma degli stati e delle transizioni

1. Progettare una tabella diversa per la rappresentazione in Java dello stato degli oggetti della classe UML *Libro*.
2. Realizzare la classe Java *Libro* tenendo conto della nuova rappresentazione.
3. Verificare che i programmi clienti realizzati per l'esercizio 5 vengano compilati ed eseguiti correttamente.

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..1, a responsabilità singola, con attributi;
- associazioni binarie, con molteplicità 0..*, a responsabilità singola;
- associazioni binarie, con molteplicità 0..1, a responsabilità doppia;

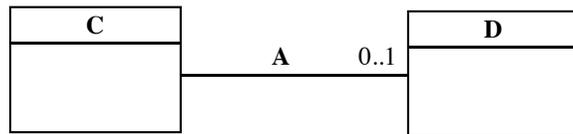
52

Realizzazione di associazioni (cont.)

- associazioni binarie, con molteplicità 0..1 o 0..*, a responsabilità doppia;
- associazioni binarie, con molteplicità massima finita;
- associazioni binarie, con molteplicità minima diversa da zero;
- associazioni n-arie;
- associazioni ordinate.

53

Realizzazione delle associazioni: primo caso



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.

54

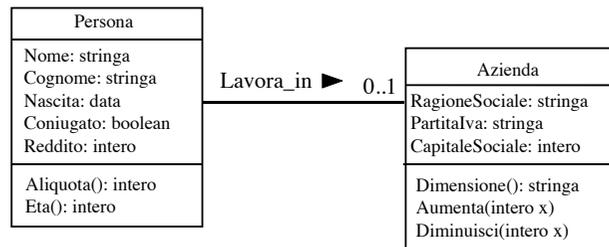
Realizzazione delle associazioni: primo caso

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

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

55

Due classi legate da associazione: esempio



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

- la ragione sociale e la partita Iva di un'azienda **non cambiano**;
- solo *Persona* abbia responsabilità sull'associazione (non ci interessa conoscere i dipendenti di un'azienda, ma solo in quale azienda lavora una persona che lavora).

56

Specifica della classe UML Azienda

InizioSpecificaClasse Azienda

Dimensione (): *stringa*

pre: nessuna

post: *result* vale "Piccola" se *this.CapitaleSociale* è inferiore a 51, vale "Media" se *this.CapitaleSociale* è compreso fra 51 e 250, vale "Grande" se *this.CapitaleSociale* è superiore a 250

Aumenta (i: intero)

pre: $i > 0$

post: *this.CapitaleSociale* vale $pre(this.CapitaleSociale) + i$

Diminuisci (i: intero)

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

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

FineSpecifica

57

Realizzazione in Java della classe 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() +
```

58

```

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

```

Realizzazione in Java della classe Persona

```

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

```

59

Esercizio 6: cliente

Realizzare in Java lo use case *Analisi Aziende*, specificato di seguito, progettando prima un opportuno algoritmo:

InizioSpecificaUseCase **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*

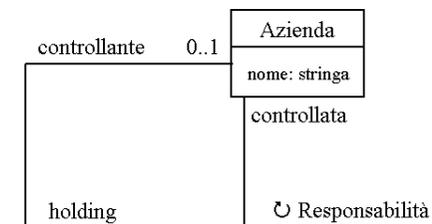
FineSpecifica

60

Realizzazione delle associazioni: secondo caso

Quando c
 la stessa
ai ruoli,

più volte
 attribuisce



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.

61

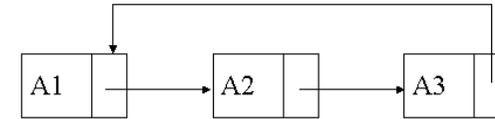
Realizzazione delle associazioni: secondo caso

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

```
// File Ruoli/Azienda.java

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

62



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".

63

Esercizio 8: cliente

Realizzare in Java lo use case *Ricognizione truffe*, specificato di seguito, progettando prima un opportuno algoritmo:

InizioSpecificaUseCase 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

64

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.

Rimangono le seguenti assunzioni:

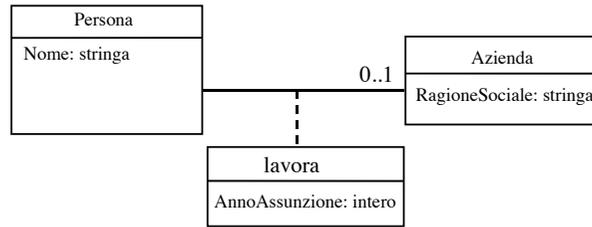
- molteplicità 0..1;
- solo una delle due classi *ha responsabilità* sull'associazione (dobbiamo rappresentare **un solo verso** dell'associazione).

Gli altri casi (altre molteplicità, responsabilità sull'associazione di entrambe le classi) verranno considerati in seguito.

65

Attributi di associazione: esempio

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



In questo caso sono possibili due strategie:

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).

66

Realizzazione delle associazioni: esempio

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() { return annoAssunzione; }
    public void setLavora(Azienda a, int x) {
        if (a != null) { lavora = a; annoAssunzione = x; }
    }
    public void eliminaLavora() { lavora = null; }
}
```

67

Osservazione sull'attributo dell'associazione

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`).

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

68

Attributi di associazione (cont.)

Consideriamo adesso la **seconda** strategia, che è quella da preferirsi.

Per rappresentare l'associazione *As* fra le classi UML *A* e *B* introduciamo **una ulteriore classe** Java `TipoLinkAs`, che ha lo scopo di rappresentare i link fra gli oggetti delle classi *A* e *B*.

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

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

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

69

Funzioni della classe Java TipoLinkAs

La classe Java `TipoLinkAs` 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 A e B 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.

70

Attributi di associazione (cont.)

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

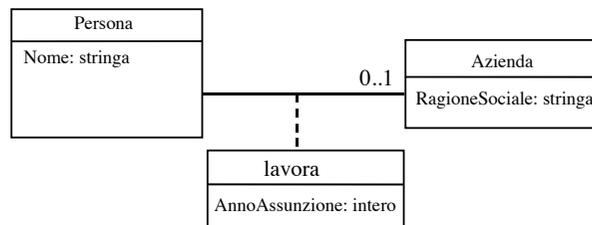
In particolare, la classe Java A avrà:

- un campo dato di tipo `TipoLinkAs`, per rappresentare l'eventuale link; in particolare, se tale campo vale `null`, allora significa che l'oggetto di classe A non è associato ad un oggetto di classe B;
- dei campi funzione che permettano di gestire il link (funzioni `get`, `inserisci`, `elimina`).

71

Attributi di associazione: esempio

Riprendiamo l'esempio, e trattiamolo con la **seconda** strategia:



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).

72

Realizzazione in Java della classe 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;
```

73

```

    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; }
}

```

Realizzazione in Java della classe 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;
    }
}

```

74

Realizzazione in Java della classe 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; }
}

```

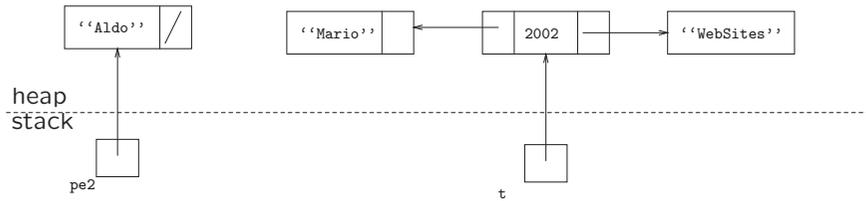
75

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()`.

76

Controllo coerenza riferimenti



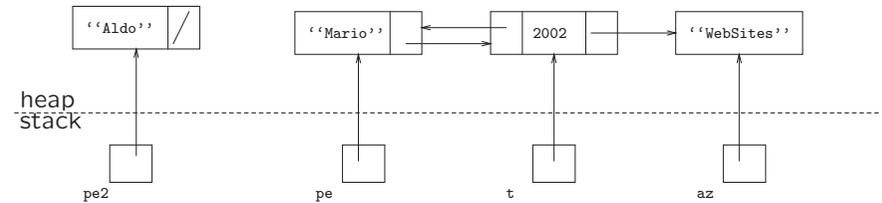
Il link `t` non si riferisce all'oggetto "Aldo".

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.

77

Possibile stato della memoria



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

78

Realizzazione della situazione di esempio

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

79

Esercizio 9: cliente

Realizzare in Java lo use case *Ristrutturazione Industriale*, specificato di seguito, progettando prima opportuni algoritmi:

InizioSpecificaUseCase Ristrutturazione Industriale

AssunzioneInBlocco (*i: Insieme(Persona), a: Azienda, an: intero*)

pre: nessuna

post: tutte le persone nell'insieme di persone *i* vengono assunte dall'azienda *a* nell'anno *an*

AssunzionePersonaleEsperto (*i: Insieme(Persona), a: Azienda, av: intero, an: intero*)

pre: $an \geq av$

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

80

Molteplicità 0..*

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

Rimangono le seguenti assunzioni:

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

Gli altri casi verranno considerati in seguito.

81

Molteplicità 0..* (cont.)

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

In particolare, la classe Java *A* avrà:

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

82

Molteplicità 0..*: 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*.

83

Realizzazione in Java della classe 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);
    }
}
```

84

```

public Set<Azienda> getLinkHaLavorato() {
    return (HashSet<Azienda>)insieme_link.clone();
}
}

```

Classe Java Persona: considerazioni

- La classe ha un campo dato di tipo `HashSet<Azienda>`.
- Il costruttore della classe `Persona` crea un oggetto di tale classe, usando 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.

85

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 dalla specifica struttura dati utilizzata per realizzare le funzionalità previste da `Set`, aumentando così l'information hiding.

86

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.

87

Esercizio 10: cliente

Realizzare in Java lo use case *Analisi del collocamento*, specificato di seguito, progettando prima un opportuno algoritmo:

InizioSpecificaUseCase **Analisi del collocamento**

PiùDipendenti (*i*: *Insieme(Persona)*): Azienda

pre: nessuna

post: *result* è l'azienda per cui più dipendenti fra le persone di *i* hanno lavorato

FineSpecifica

88

Esercizio 11: schema realizzativo con condivisione

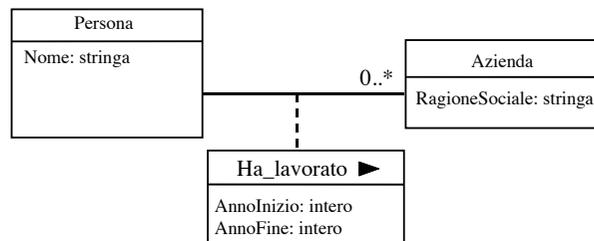
Realizzare in Java la classe *Persona* mediante lo schema realizzativo con condivisione di memoria e lo use case *Analisi del collocamento* che fa uso di tale versione della classe.

89

Combinazione: 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):



90

Combinazione (cont.)

Dobbiamo combinare le scelte fatte in precedenza:

1. come per tutte le associazioni con attributi, dobbiamo definire una apposita classe Java per la rappresentazione del link (*TipoLinkHaLavorato*); inoltre, dobbiamo prevedere la possibilità che il costruttore di questa classe lanci un'eccezione nel caso in cui i riferimenti passatigli come argomento siano pari a `null`;
2. come per tutte le associazioni con vincolo di molteplicità 0..*, dobbiamo utilizzare una struttura di dati per la rappresentazione dei link.

91

Combinazione: 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.

92

```
        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; }
}
```

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;

```

93

Combinazione: la classe Persona

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

Scegliamo ancora di utilizzare 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.

94

La 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)
```

95

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 (II strategia)	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

96

```
        insieme_link.remove(t);
    }
    public Set<TipoLinkHaLavorato> getLinkHaLavorato() {
        return (HashSet<TipoLinkHaLavorato>)insieme_link.clone();
    }
}
```

Esercizio 12: cliente

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

InizioSpecificaUseCase Analisi Mercato Lavoro

PeriodoPiùLungo (*p: Persona*): intero

pre: nessuna

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

RiAssuntoSubito (*p: Persona*): booleano

pre: nessuna

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

SonoStatiColleghi (*p1: Persona, p2: Persona*): booleano

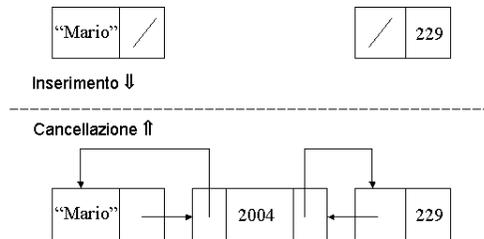
pre: nessuna

post: *result* vale *true* se e solo se *p1* e *p2* hanno lavorato contemporaneamente per la stessa azienda

FineSpecifica

97

Mantenimento coerenza



102

Resp. di entrambe le classi UML (cont.)

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

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

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

Continuiamo ad utilizzare (come in tutti i casi in cui c'è necessità di rappresentare attributi di associazione) una classe Java per i link, in questo caso *TipoLinkOccupazione*, che modella tuple del prodotto cartesiano tra *Stanza* e *Persona* con attributo *DaAnno*.

103

Caratteristiche delle classi Java

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

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

Stanza: del tutto simile a *Persona*.

104

La classe Java Persona

```
// File RespEntrambi01/Persona.java
public class Persona {
    private final String nome;
    private TipoLinkOccupazione link;
    public Persona(String n) { nome = n; }
    public String getNome() { return nome; }
    public void inserisciLinkOccupazione(AssociazioneOccupazione a) {
        if (a != null) link = a.getLink();
    }
    public void eliminaLinkOccupazione(AssociazioneOccupazione a) {
        if (a != null) link = null;
    }
    public TipoLinkOccupazione getLinkOccupazione() {
        return link;
    }
}
```

105

La classe Java Stanza

```
// File RespEntrambi01/Stanza.java
public class Stanza {
    private final int numero;
    private TipoLinkOccupazione link;
    public Stanza(int n) { numero = n; }
    public int getNumero() { return numero; }
    public void inserisciLinkOccupazione(AssociazioneOccupazione a) {
        if (a != null) link = a.getLink();
    }
    public void eliminaLinkOccupazione(AssociazioneOccupazione a) {
        if (a != null) link = null;
    }
    public TipoLinkOccupazione getLinkOccupazione() {
        return link;
    }
}
```

106

Caratteristiche delle classi Java (cont.)

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

- tre campi dati (per la stanza, per la persona e per l'attributo dell'associazione);
- un costruttore, che inizializza questi campi utilizzando i suoi argomenti; lancia un'eccezione se i riferimenti alla stanza o alla persona sono null;
- tre funzioni get, per interrogare l'oggetto;
- la funzione equals per verificare l'uguaglianza solo sugli oggetti collegati dal link, ignorando gli attributi;
- la funzione hashCode, ridefinita di conseguenza.

107

La classe Java TipoLinkOccupazione

```
// File RespEntrambi01/TipoLinkOccupazione.java

public class TipoLinkOccupazione {
    private final Stanza laStanza;
    private final Persona laPersona;
    private final int daAnno;
    public TipoLinkOccupazione(Stanza x, Persona y, int a)
        throws EccezionePrecondizioni {
        if (x == null || y == null) // CONTROLLO PRECONDIZIONI
            throw new EccezionePrecondizioni
                ("Gli oggetti devono essere inizializzati");
        laStanza = x; laPersona = y; daAnno = a;
    }
    public boolean equals(Object o) {
        if (o != null && getClass().equals(o.getClass())) {
            TipoLinkOccupazione b = (TipoLinkOccupazione)o;
```

108

```
        return b.laPersona == laPersona && b.laStanza == laStanza;
    }
    else return false;
}
public int hashCode() {
    return laPersona.hashCode() + laStanza.hashCode();
}
public Stanza getStanza() { return laStanza; }
public Persona getPersona() { return laPersona; }
public int getDaAnno() { return daAnno; }
}
```

Caratteristiche delle classi Java (cont.)

AssociazioneOccupazione: avrà:

- un campo dato, di tipo `TipoLinkOccupazione` per la rappresentazione del link;
- funzioni per la gestione di questo campo, in particolare:
 - `static void inserisci(TipoLinkOccupazione)`, per associare un link fra una persona ed una stanza;
 - `static void elimina(TipoLinkOccupazione)`, per rimuovere un link fra una persona ed una stanza;
 - `TipoLinkOccupazione getLink()`, per ottenere il link;
- il costruttore sarà **privato**;
- la classe sarà **final**, per evitare che si possa definire una sottoclasse in cui il costruttore è pubblico.

109

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

La classe Java AssociazioneOccupazione

// File RespEntrambi01/AssociazioneOccupazione.java

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

110

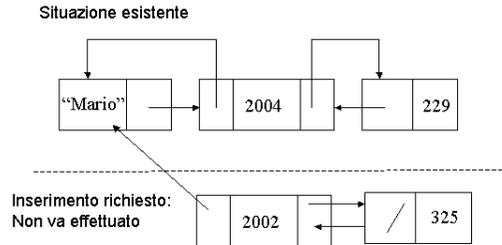
Inserimento di link: controlli

Si noti che è necessario prevenire la possibilità di richiedere agli oggetti di tipo `Stanza` o `Persona` di inserire link quando gli oggetti sono già “occupati” (creando in questa maniera situazioni inconsistenti).

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

111

Inserimento di link: esempio



112

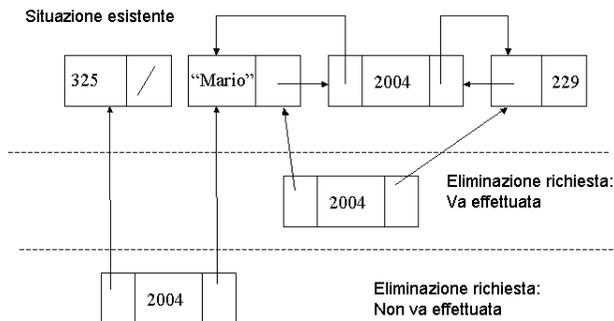
Eliminazione di link: controlli

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

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

113

Eliminazione di link: esempio



114

La classe Java *AssociazioneOccupazione* (cont.)

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

I clienti saranno in grado di:

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

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

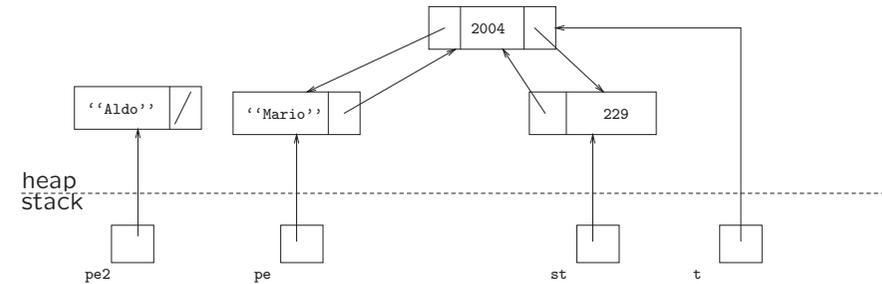
115

Considerazioni sulle classi Java

- Le funzioni `inserisciLinkOccupazione()` ed `eliminaLinkOccupazione()` della classe `Persona` di fatto possono essere invocate **solamente dalla classe** `AssociazioneOccupazione`, in quanto:
 - per invocarle dobbiamo passare loro degli argomenti di tipo `AssociazioneOccupazione`, e
 - gli oggetti della classe `AssociazioneOccupazione` non possono essere creati, se non attraverso le funzioni `inserisci()` ed `elimina()` di quest'ultima.
- Forziamo quindi i clienti che vogliono stabilire o rimuovere dei link ad usare le funzioni (statiche, in quanto svincolate da oggetti di invocazione) `inserisci()` ed `elimina()` della classe `AssociazioneOccupazione`, e non effettuare inserimenti e cancellazioni di link direttamente mediante le classi `Persona` e `Stanza`.

116

Possibile stato della memoria



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

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

117

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);
}
AssociazioneOccupazione.inserisci(t);
```

118

Esercizio 13: cliente

Realizzare in Java lo use case *Riallocazione Personale*:

InizioSpecificaUseCase Riallocazione Personale

Promuovi (*ins: Insieme(Persona), st: Stanza, anno: intero*)

pre: *ins* non è vuoto; almeno ad una persona di *ins* è assegnata una stanza

post: ad una delle persone di *ins* che sono da più tempo nella stessa stanza viene assegnata la stanza *st*, a partire dall'anno *anno*

Libera (*ins: Insieme(Stanza)*)

pre: a tutte le stanze di *ins* è assegnata una persona

post: le stanze di *ins* che sono occupate da più tempo vengono liberate

...

119

Esercizio 13: cliente (cont.)

...

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

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

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

FineSpecifica

120

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

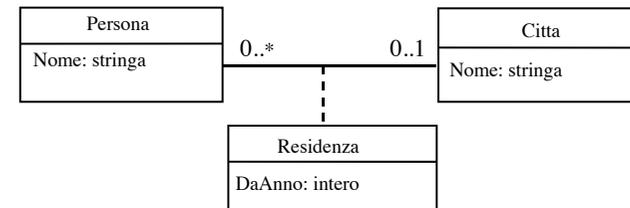
La metodologia proposta per la molteplicità 0..1 può essere usata anche per la molteplicità 0..* (per il momento, una delle due molteplicità è ancora 0..1). Le differenze principali sono le seguenti:

- La classe Java (nel nostro esempio: *Citta*) i cui oggetti possono essere legati a più oggetti dell'altra classe Java (nel nostro esempio: *Persona*) ha le seguenti caratteristiche:
 - ha un ulteriore campo dato di tipo *Set*, per poter rappresentare tutti i link;
 - l'oggetto di classe *Set* viene creato tramite il costruttore;
 - ha tre campi funzione (*inserisciLinkResidenza()*, *eliminaLinkResidenza()* e *getLinkResidenza()*) per la gestione dell'insieme dei link; quest'ultima restituisce **una copia** dell'insieme dei link.

122

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

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



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

121

La classe Java Citta

```
// File RespEntrambi0STAR/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(AssociazioneResidenza a) {
        if (a != null) insieme_link.add(a.getLink());
    }
    public void eliminaLinkResidenza(AssociazioneResidenza a) {
        if (a != null) insieme_link.remove(a.getLink());
    }
}
```

123

```

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.
- 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.

124

La classe Java Persona

// File RespEntrambiOSTAR/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(AssociazioneResidenza a) {
        if (a != null) link = a.getLink();
    }
    public void eliminaLinkResidenza(AssociazioneResidenza a) {
        if (a != null) link = null;
    }
    public TipoLinkResidenza getLinkResidenza() {
        return link;
    }
}

```

125

La classe Java TipoLinkResidenza

// File RespEntrambiOSTAR/TipoLinkResidenza.java

```

public class TipoLinkResidenza {
    private final Citta laCitta;
    private final Persona laPersona;
    private final int daAnno;
    public TipoLinkResidenza(Citta x, Persona y, int a)
        throws EccezionePrecondizioni {
        if (x == null || y == null) // CONTROLLO PRECONDIZIONI
            throw new EccezionePrecondizioni
                ("Gli oggetti devono essere inizializzati");
        laCitta = x; laPersona = y; daAnno = a;
    }
    public boolean equals(Object o) {
        if (o != null && getClass().equals(o.getClass())) {
            TipoLinkResidenza b = (TipoLinkResidenza)o;

```

126

```

        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; }
}

```

```

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

```

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

La classe Java AssociazioneResidenza

```

// File RespEntrambiOSTAR/AssociazioneResidenza.java

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

```

127

Esercizio 14: cliente

Realizzare in Java lo use case *Gestione Anagrafe*:

InizioSpecificaUseCase **Gestione Anagrafe**

TrovaNuovi (*c*: Città, *a*: intero): *Insieme(Persona)*

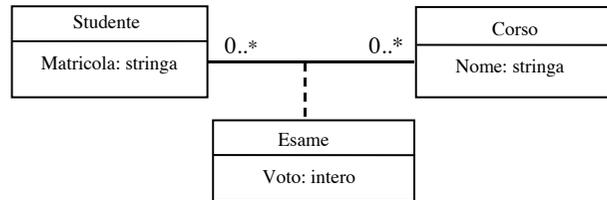
pre: nessuna

post: *result* è l'insieme di persone che sono residenti nella città *c* da non prima dell'anno *a*

FineSpecifica

Entrambe le molteplicità sono 0..*

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



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

129

La classe Java Studente

```
// File RespEntrambi0STAR2/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(AssociazioneEsame a) {
        if (a != null) insieme_link.add(a.getLink());
    }
    public void eliminaLinkEsame(AssociazioneEsame a) {
        if (a != null) insieme_link.remove(a.getLink());
    }
}
```

131

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

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

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

- hanno un ulteriore campo dato di tipo `HashSet<TipoLinkEsame>`, per poter rappresentare tutti i link;
l'oggetto di classe `HashSet<TipoLinkEsame>` viene creato tramite il costruttore;
- hanno tre campi funzione (`inserisciLinkResidenza()`, `eliminaLinkResidenza()` e `getLinkResidenza()`) per la gestione dell'insieme dei link;
quest'ultima restituisce **una copia** dell'insieme dei link.

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

```
public Set<TipoLinkEsame> getLinkEsame() {
    return (HashSet<TipoLinkEsame>)insieme_link.clone();
}
}
```

130

La classe Java Corso

```
// File RespEntrambi0STAR2/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(AssociazioneEsame a) {
        if (a != null) insieme_link.add(a.getLink());
    }
    public void eliminaLinkEsame(AssociazioneEsame a) {
        if (a != null) insieme_link.remove(a.getLink());
    }
}
```

132

La classe Java TipoLinkEsame

```
// File RespEntrambi0STAR2/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;
```

133

```
        public Set<TipoLinkEsame> getLinkEsame() {
            return (HashSet<TipoLinkEsame>)insieme_link.clone();
        }
    }
}

        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; }
}
```

La classe Java AssociazioneEsame

```
// File RespEntrambi0STAR2/AssociazioneEsame.java

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

134

Esercizio 15: cliente

Realizzare in Java lo use case *Valutazione Didattica*:

InizioSpecificaUseCase Valutazione Didattica

StudenteBravo (*s: Studente*): *booleano*

pre: nessuna

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

CorsoFacile (*c: Corso*): *booleano*

pre: nessuna

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

FineSpecifica

135

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

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

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 **devono poter essere interrogati** sul numero di link esistenti.

136

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 .

137

Considerazioni sulla molteplicità

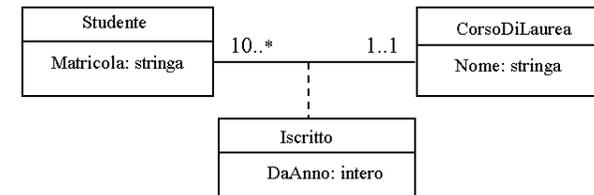
Questo esempio dimostra bene il fatto che imporre che tutti i vincoli di molteplicità di un diagramma delle classi siano rispettati *in ogni momento* è, in generale, molto complicato.

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 .

139

Esempio con molteplicità arbitrarie



Prenderemo in considerazione questo esempio.

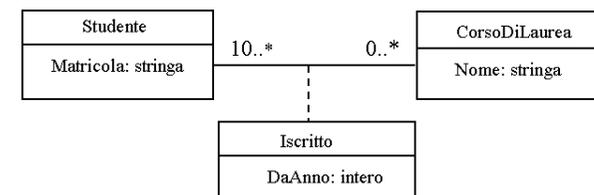
138

Molteplicità minima diversa da zero

Per esige

Consideri
classi (si

ma delle



Supponiamo che sia $Studiante$ sia $CorsoDiLaurea$ abbiano responsabilità sull'associazione.

140

Molteplicità minima diversa da zero (cont.)

- Rispetto al caso di associazione con responsabilità doppia e in cui i vincoli di molteplicità siano entrambi 0..*, la classe Java `CorsoDiLaurea` si differenzia nei seguenti aspetti:
 1. Ha un'ulteriore funzione pubblica `int quantiIscritti()`, che restituisce il numero di studenti iscritti per il corso di laurea oggetto di invocazione.
In questa maniera, il cliente si può rendere conto se il vincolo di molteplicità sia rispettato oppure no.
 2. La funzione `int getLinkIscritto()` lancia una opportuna eccezione (di tipo `EccezioneCardMin`) quando l'oggetto di invocazione non rispetta il vincolo di cardinalità minima sull'associazione `Iscritto`.

141

La 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;
    }
}
```

143

Molteplicità minima diversa da zero (cont.)

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

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

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

142

La 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(AssociazioneIscritto a) {
```

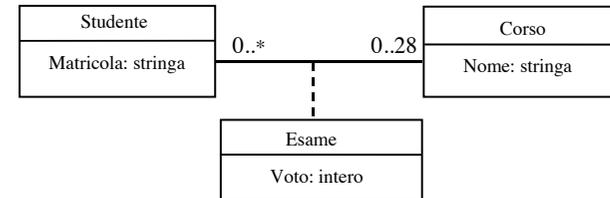
144

```

    if (a != null) insieme_link.add(a.getLink());
}
public void eliminaLinkIscritto(AssociazioneIscritto a) {
    if (a != null) insieme_link.remove(a.getLink());
}
public Set<TipoLinkIscritto> getLinkIscritto() throws EccezioneCardMin {
    if (quantiIscritti() < MIN_LINK_ISCRITTO)
        throw new EccezioneCardMin("Cardinalita' minima violata");
    else return (HashSet<TipoLinkIscritto>)insieme_link.clone();
}
}
}

```

Molteplicità massima finita



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

145

Molteplicità massima finita (cont.)

- Rispetto al caso di associazione con responsabilità doppia e in cui i vincoli di molteplicità siano entrambi 0..*, la classe Java *Studente* si differenzia nei seguenti aspetti:

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

146

Molteplicità massima finita (cont.)

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

- *Corso*,
- *TipoLinkEsame*,
- *EccezionePrecondizioni*.
- *AssociazioneEsame*,

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

147

La 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;
    }
}

}

public void inserisciLinkEsame(AssociazioneEsame a) {
    if (a != null) insieme_link.add(a.getLink());
}

public void eliminaLinkEsame(AssociazioneEsame a) {
    if (a != null) insieme_link.remove(a.getLink());
}

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

148

La classe Java Studente

```
// File MoltMax/Studente.java

import java.util.*;

public class Studente {
    private final String matricola;
    private HashSet<TipoLinkEsame> insieme_link;
    public static final int MAX_LINK_ESAME = 28;
    // PER IL VINCOLO DI MOLTEPLICITÀ
    public Studente(String n) {
        matricola = n;
        insieme_link = new HashSet<TipoLinkEsame>();
    }
    public String getMatricola() { return matricola; }
    public int quantiEsami() { // FUNZIONE NUOVA
        return insieme_link.size();
    }
}
```

149

Esercizio 16

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

150

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.

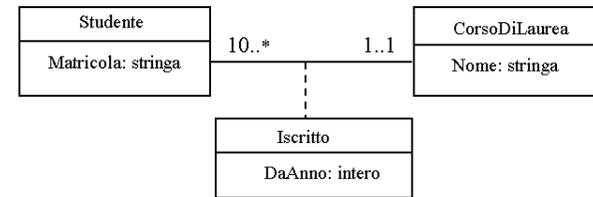
151

La classe Java Studente

```
// File MoltMax1/Studente.java
public class Studente {
    private final String matricola;
    private TipoLinkIscritto link;
    public static final int MIN_LINK_ISCRITTO = 1;
    public Studente(String n) { matricola = n; }
    public String getMatricola() { return matricola; }
    public int quantiIscritti() { // FUNZIONE NUOVA
        if (link == null)
            return 0;
        else return 1;
    }
    public void inserisciLinkIscritto(AssociazioneIscritto a) {
        if (a != null) link = a.getLink();
    }
    public void eliminaLinkIscritto(AssociazioneIscritto a) {
```

153

Molteplicità massima 1 (cont.)



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

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

152

```
        if (a != null) link = null;
    }
    public TipoLinkIscritto getLinkIscritto() throws EccezioneCardMin {
        if (link == null)
            throw new EccezioneCardMin("Cardinalita' minima violata");
        else
            return link;
    }
}
```

La classe Java AssociazioneIscritto

```
// File MoltMax1/AssociazioneIscritto.java
public final class AssociazioneIscritto {
    private AssociazioneIscritto(TipoLinkIscritto x) { link = x; }
    private TipoLinkIscritto link;
    public TipoLinkIscritto getLink() { return link; }
    public static void inserisci(TipoLinkIscritto y) {
        if (y != null && y.getStudiante().quantiIscritti() == 0) {
            AssociazioneIscritto k = new AssociazioneIscritto(y);
            k.link.getCorsoDiLaurea().inserisciLinkIscritto(k);
            k.link.getStudiante().inserisciLinkIscritto(k);
        }
    }
    public static void elimina(TipoLinkIscritto y) {
        try {
            if (y != null && y.getStudiante().getLinkIscritto().equals(y) ) {
                AssociazioneIscritto k = new AssociazioneIscritto(y);
```

154

Molteplicità massima 1 (cont.)

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

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

Rimangono invece inalterate le seguenti classi Java:

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

155

```
        k.link.getCorsoDiLaurea().eliminaLinkIscritto(k);
        k.link.getStudiante().eliminaLinkIscritto(k);
    }
}
catch (EccezioneCardMin e) {
    System.out.println(e);
}
}
}
```

Esercizio 17: cliente

Realizzare in Java lo use case *Cambiamento Indirizzo*:

InizioSpecificaUseCase Cambiamento Indirizzo

Cambiano (*i*: *Insieme(Studiante)*, *c*: *CorsoDiLaurea*, *a*: *intero*)

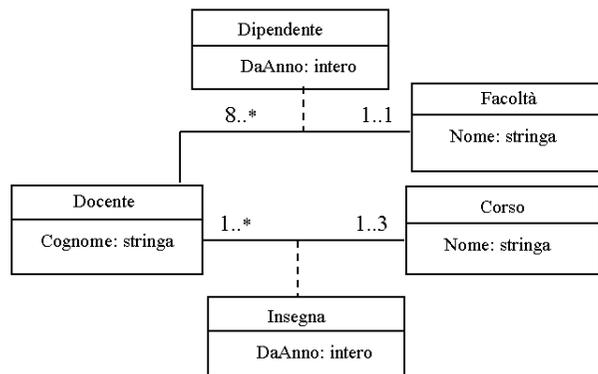
pre: nessuna

post: ogni studente di *i* si iscrive al corso di laurea *c* a partire dall'anno *a*

FineSpecifica

156

Esercizio 18



Realizzare in Java questo diagramma delle classi.

157

Associazioni n-arie

Si trattano generalizzando quanto visto per le associazioni binarie.

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

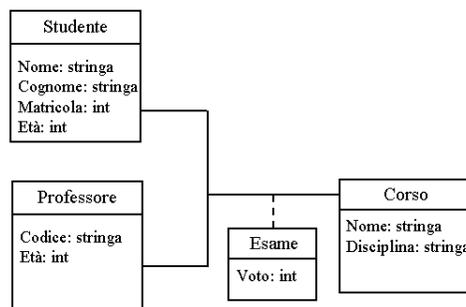
In ogni caso, per un'associazione n-aria *A*, anche se non ha attributi, si definisce la corrispondente classe *TipoLinkA*.

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

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

158

Esempio



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

159

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;`
 - `AssociazioneOccupazione;`
 - `Studente/Professore;`
`inserisci()/elimina(),`
 - `Corso.`

160

Esercizio 19

1. Realizzare tutte le classi Java per l'esempio corrente.
2. Con riferimento all'esercizio 15, realizzare in Java l'operazione *StudenteBravo* dello use case *Valutazione Didattica* per il caso corrente.
3. Arricchire lo use case con la seguente operazione, e realizzarla in Java:

ProfessoreBuono (*p: Professore*): *booleano*

pre: nessuna

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

161

Esercizio 21

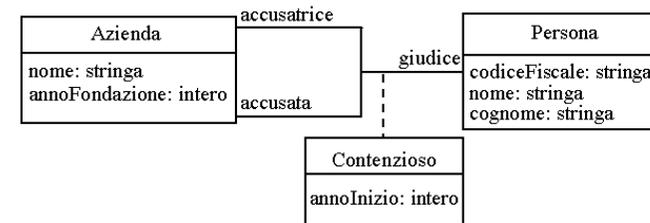
Con riferimento all'esercizio precedente, fornire la specifica, gli algoritmi e la realizzazione di questo use case.

L'ufficio giudiziario del lavoro deve poter effettuare, come cliente della nostra applicazione, dei controlli sui contenziosi. A questo scopo, si faccia riferimento ad uno use case che prevede le seguenti operazioni:

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

163

Esercizio 20



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

162

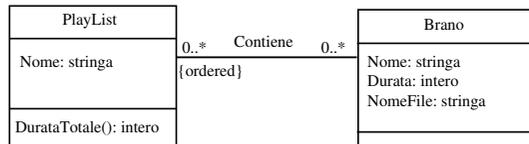
Associazioni Ordinate

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

164

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).

165

Specifica della classe UML Playlist

InizioSpecificazione Classe Playlist

durataTotale (): intero

pre: nessuna

post: *result* è pari alla somma delle durate dei brani contenuti in *this*

FineSpecificazione

166

Realizzazione in Java della classe Playlist

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

167

```
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; }
}
```

168

Realizzazione in Java del cliente

La progettazione dell'algoritmo è lasciata come esercizio

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

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

170

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

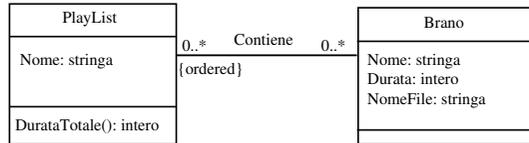
post: *result* è costituito dalle PlayList di *i* la cui durata totale è massima

FineSpecifica

```
        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).

171

Esempio responsabilità doppia (cont.)

- In questo caso dobbiamo adattare la metodologia generale, prevedendo:
 - la realizzazione della classe `TipoLinkContiene`,
 - la realizzazione della classe `AssociazioneContiene`,
 - 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.

172

Realizzazione in Java della classe Playlist

```
// File OrdinateEntrambiOSTAR/Playlist.java
import java.util.*;
public class Playlist {
    private final String nome;
    private LinkedList<TipoLinkContiene> insieme_link;
    public Playlist(String n) {
        nome = n;
        insieme_link = new LinkedList<TipoLinkContiene>();
    }
    public String getNome() { return nome; }
    public void inserisciLinkContiene(AssociazioneContiene a){
        if (a != null && !insieme_link.contains(a.getLink()))
            insieme_link.add(a.getLink());
    }
    public void eliminaLinkContiene(AssociazioneContiene a) {
        if (a != null) insieme_link.remove(a.getLink());
    }
}
```

173

```
}
public List<TipoLinkContiene> getLinkContiene() {
    return (LinkedList<TipoLinkContiene>)insieme_link.clone();
}
public int durataTotale() {
    int result = 0;
    Iterator<TipoLinkContiene> il = insieme_link.iterator();
    while (il.hasNext()) {
        Brano b = il.next().getBrano();
        result = result + b.getDurata();
    }
    return result;
}
}
```

Realizzazione in Java della classe Brano

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

174

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

Realizzazione della classe TipoLinkContiene

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

175

Realizzazione della classe AssociazioneContiene

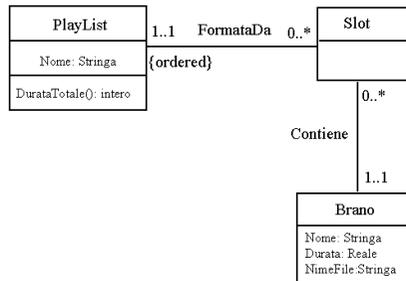
```
// File OrdinateEntrambiOSTAR/AssociazioneContiene.java
public final class AssociazioneContiene {
    private AssociazioneContiene(TipoLinkContiene x) { link = x; }
    private TipoLinkContiene link;
    public TipoLinkContiene getLink() { return link; }
    public static void inserisci(TipoLinkContiene y) {
        if (y != null) {
            AssociazioneContiene k = new AssociazioneContiene(y);
            y.getPlaylist().inserisciLinkContiene(k);
            y.getBranO().inserisciLinkContiene(k);
        }
    }
    public static void elimina(TipoLinkContiene y) {
        if (y != null) {
            AssociazioneContiene k = new AssociazioneContiene(y);
            y.getPlaylist().eliminaLinkContiene(k);
            y.getBranO().eliminaLinkContiene(k);
        }
    }
}
}
```

176

Esercizio 22

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

- solo **Slot** ha responsabilità su **Contiene** (cioè non siamo interessati a sapere
- sia *PlayL*



177

Realizzazione di generalizzazioni

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

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

179

Esercizio 22 (cont.)

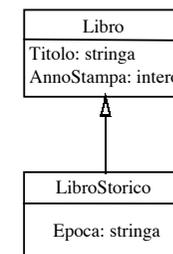
Nella realizzazione, possiamo tenere conto che (cfr. *Seconda Parte*) la classe **Slot** non è presente esplicitamente nei requisiti, ed è stata introdotta per poter distinguere la posizione dei brani dai brani stessi.

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

178

Generalizzazione

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



180

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*.

181

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).

183

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*).

182

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.

184

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`.

185

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).

187

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 PACKAGE	SI	SI	SI	NO	2	
CLASSE DERIVATA PACKAGE DIVERSO	SI	SI	NO	NO	3	
CL. NON DERIVATA PACKAGE DIVERSO	SI	NO	NO	NO	4	
					V	
					V	

NOTA: Decrescono i diritti

186

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.*;
```

188

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;
    }
}
```

189

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;
    }
}
```

191

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).

190

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.

192

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() { }
```

193

Riassunto struttura file e direttori

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

194

Ridefinizione

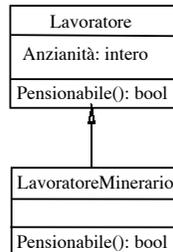
Nella classe derivata è possibile fare **overriding** (dall'inglese, *ridefinizione*, *sovrascrittura*) delle funzioni della classe base.

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

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

195

Ridefinizione: esempio



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

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

196

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; }
}
```

197

Ridefinizione: classe derivata

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

package LavoratoreMinerario;
import Lavoratore.*;

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

198

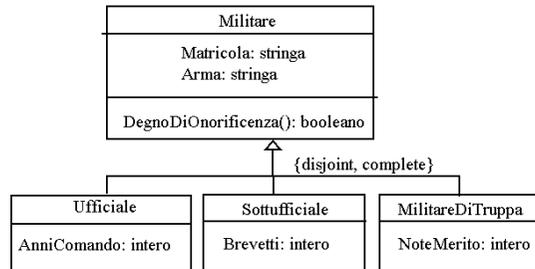
Generalizzazioni disgiunte e complete

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.

199

Generalizzazioni disgiunte e complete (cont.)



200

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*.

202

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).

201

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*.

203

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.

204

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");
}
```

206

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;
    }
}
```

205

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;
    }
}
```

207

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;
    }
}
```

208

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;
    }
}
```

209

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`.

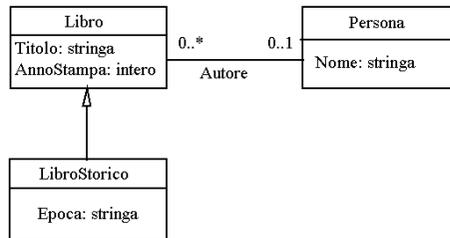
210

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.

211

Packages, esempio



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

212

Struttura file e direttori

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

214

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

213

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 ;
    }
}
```

215

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;
    }
}
```

216

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;
    }
}
```

217

La classe Java StatisticaAutori

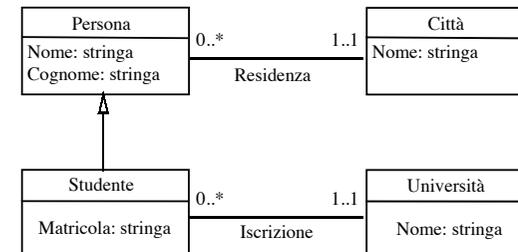
```
// File PackageLibri/StatisticaAutori.java

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() { }
}
```

218

Esercizio 23



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

219

Esercizio 24

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

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

220

Aspetti di UML non trattati

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

- diagrammi degli use case con inclusioni, estensioni e generalizzazioni;
- generalizzazioni non disgiunte;
- specializzazione di attributi;
- specializzazione di associazioni;
- ereditarietà multipla.

221

Soluzione esercizio 1

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

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

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

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

Soluzioni degli esercizi della quarta parte

222

223

```

        return true;
    }
    return false;
}
private AnalisiRecapiti() {};
}

```

```

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

```

Soluzione esercizio 2

```

// File MolteplicitaAttributiCond/GestioneRubrica.java
import java.util.*;
import IteratoreSolaLetture.*;

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

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

```

224

Soluzione esercizio 3

```

// File AttributiEOperazioni/Esercizio/Persona.java

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

```

225

```

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

```

```

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

```

[Soluzione esercizio 4](#)

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

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

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

```

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

```

```

                aliquotaMassima = elem.aliquota();
        }
        int quantiRicchi = 0;
        double sommaEtaRicchi = 0.0;
        it = i.iterator();
        while(it.hasNext()) {
            Persona elem = it.next();
            if (elem.aliquota() == aliquotaMassima) {
                sommaEtaRicchi += elem.eta(g,m,a);
                quantiRicchi++;
            }
        }
        return sommaEtaRicchi / quantiRicchi;
    }
    private AnalisiRedditi() {}
}

```

Soluzione esercizio 5

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

```
per ogni libro elem di i
    se !elem.cancellatoDalCatalogo
        allora Libro libroPiuVecchio = elem
            int annoPiuVecchio = libroPiuVecchio.annoPubblicazione;
per ogni altro libro elem di i
    se !elem.cancellatoDalCatalogo &&
    elem.annoPubblicazione < annoPiuVecchio
        allora libroPiuVecchio = elem
            annoPiuVecchio = libroPiuVecchio.annoPubblicazione;
return libroPiuVecchio;
```

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

```
int quantiInCatalogo = 0;
int etaTotaleInCatalogo = 0;
per ogni libro elem di i
    se !elem.cancellatoDalCatalogo
        allora quantiInCatalogo++;
        etaTotaleInCatalogo += a - elem.getAnnoPubblicazione();
return etaTotaleInCatalogo/quantiInCatalogo;
```

227

```
        elem.getAnnoPubblicazione() < annoPiuVecchio) {
            libroPiuVecchio = elem;
            annoPiuVecchio = elem.getAnnoPubblicazione();
        }
    }
    return libroPiuVecchio;
}
public static double etaMediaInCatalogo(Set<Libro> i, int a) {
    int quantiInCatalogo = 0;
    int etaTotaleInCatalogo = 0;
    Iterator<Libro> it = i.iterator();
    while(it.hasNext()) {
        Libro elem = it.next();
        if (!elem.cancellatoDalCatalogo()) {
            quantiInCatalogo++;
            etaTotaleInCatalogo += a - elem.getAnnoPubblicazione();
        }
    }
    return (double)etaTotaleInCatalogo/quantiInCatalogo;
}
private ValutazioneBiblioteca() {}
}
```

Soluzione esercizio 5 (cont.)

Gli algoritmi vengono realizzati tramite le funzioni piuVecchioInCatalogo() ed etaMediaInCatalogo() della seguente classe Java.

```
// File StatiTransizioni/ValutazioneBiblioteca.java
import java.util.*;

public final class ValutazioneBiblioteca {
    public static Libro piuVecchioInCatalogo(Set<Libro> i) {
        int annoPiuVecchio = 0;
        Libro libroPiuVecchio = null;
        boolean trovato_primo = false;
        Iterator<Libro> it = i.iterator();
        while(it.hasNext() && !trovato_primo) {
            Libro elem = it.next();
            if (!elem.cancellatoDalCatalogo()) {
                libroPiuVecchio = elem;
                annoPiuVecchio = elem.getAnnoPubblicazione();
                trovato_primo = true;
            }
        }
        while(it.hasNext()) {
            Libro elem = it.next();
            if (!elem.cancellatoDalCatalogo() &&
```

228

Soluzione esercizio 6

Stato	Rappresentazione in Java				
	tipo var.	boolean	boolean	boolean	boolean
	nome var.	acquisito	in_prestito	in_ritardo	finale
acquisito	valore	true	false	false	false
in prestito	valore	false	true	false	false
in ritardo	valore	false	false	true	false
finale	valore	false	false	false	true

// File StatiTransizioni/Esercizio/Libro.java

```
public class Libro {
    private boolean acquisito = true, in_prestito = false,
        in_ritardo = false, finale = false;
    private final String titolo;
    private final int anno_pubblicazione;
    public Libro(String t, int a) {
        titolo = t;
        anno_pubblicazione = a;
    }
    public String getTitolo() { return titolo; }
    public int getAnnoPubblicazione() { return anno_pubblicazione; }
```

229

```

public void prestito() {
    if (acquisito) {
        in_prestito = true;
        acquisito = false;
    }
}
public void scadenzaTermini() {
    if (in_prestito) {
        in_ritardo = true;
        in_prestito = false;
    }
}
public void restituzione() {
    if (in_prestito || in_ritardo) {
        acquisito = true;
        in_prestito = in_ritardo = false;
    }
}
public void sollecito() {
    if (in_ritardo)
        invioLettera();
}
public void cancellazioneCatalogo() {
    finale = true;
    in_prestito = in_ritardo = acquisito = false;
}
}

```

```

public boolean cancellatoDalCatalogo() {
    return finale;
}
private void invioLettera() {
    System.out.println
        ("Gentile Utente, e' invitato a restituire il libro " + titolo);
}
}

```

Soluzione esercizio 7

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

```

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

```

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

```

// File Associazioni01/AnalisiAziende.java

import java.util.*;

public final class AnalisiAziende {
    public static double redditoMedioInGrandiAziende (Set<Persona> i) {
        int quantiInGrandiAziende = 0;

```

```

        double sommaRedditoDipendentiGrandiAziende = 0.0;
        Iterator<Persona> it = i.iterator();
        while(it.hasNext()) {
            Persona elem = it.next();
            if (elem.getLavoraIn() != null &&
                elem.getLavoraIn().dimensione().equals("Grande")) {
                quantiInGrandiAziende++;
                sommaRedditoDipendentiGrandiAziende += elem.getReddito();
            }
        }
        return sommaRedditoDipendentiGrandiAziende / quantiInGrandiAziende;
    }
    private AnalisiAziende() { }
}

```

Soluzione esercizio 8

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

ricorsivo:

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

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

```
// File Ruoli/RicognizioneTruffe.java
```

```
import java.util.*;

public final class RicognizioneTruffe {
```

231

Soluzione esercizio 9

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

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

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

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

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

```
// File Ass01Attr/RistrutturazioneIndustriale.java
```

```
import java.util.*;
```

232

```
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() { }
```

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

Soluzione esercizio 10

```
        TipoLinkLavora temp = null;
        try {
            temp = new TipoLinkLavora(az,elem,an);
        }
        catch (EccezionePrecondizioni e) {
            System.out.println(e);
        }
        elem.inserisciLinkLavora(temp);
    }
}
private RistrutturazioneIndustriale() { }
}
```

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

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

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

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

```
import java.util.*;
```

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

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

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

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

Soluzione esercizio 11

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

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

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

```
import java.util.*;

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

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

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

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

Soluzione esercizio 12

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

Soluzione esercizio 12 (cont.)

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

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

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

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

public final class AnalisiMercatoLavoro {
```

Per l'operazione **PeriodoPiuLungo** 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;
```

```
public static int periodoPiuLungo(Persona p) {
    int max = 0;
    Set<TipoLinkHaLavorato> temp = p.getLinkHaLavorato();
    Iterator<TipoLinkHaLavorato> it = temp.iterator();
    while(it.hasNext()) {
        TipoLinkHaLavorato lnk = it.next();
        int durata = lnk.getAnnoFine() - lnk.getAnnoInizio() + 1;
        if (durata > max)
            max = durata;
    }
    return max;
}

public static boolean riAssuntoSubito(Persona p) {
    Set<TipoLinkHaLavorato> temp = p.getLinkHaLavorato();
    Iterator<TipoLinkHaLavorato> it = temp.iterator();
    while(it.hasNext()) {
        TipoLinkHaLavorato lnk = it.next();
        Iterator<TipoLinkHaLavorato> it2 = temp.iterator();
        while(it2.hasNext()) {
            TipoLinkHaLavorato lnk2 = it2.next();
            if (lnk.getAnnoFine() == lnk2.getAnnoInizio() - 1)
                return true;
        }
    }
    return false;
}
```

```

public static boolean sonoStatiColleghi(Persona p1, Persona p2) {
    Set<TipoLinkHaLavorato> lavori_p1 = p1.getLinkHaLavorato();
    Set<TipoLinkHaLavorato> lavori_p2 = p2.getLinkHaLavorato();
    Iterator<TipoLinkHaLavorato> it = lavori_p1.iterator();
    while(it.hasNext()) {
        TipoLinkHaLavorato lnk = it.next();
        if (contieneCompatibile(lnk,lavori_p2))
            return true;
    }
    return false;
}
private static boolean contieneCompatibile
(TipoLinkHaLavorato t, Set<TipoLinkHaLavorato> ins) {
    // funzione di servizio: verifica se nell'insieme di link ins
    // sia presente un link compatibile con t
    Iterator<TipoLinkHaLavorato> it = ins.iterator();
    while(it.hasNext()) {
        TipoLinkHaLavorato lnk = it.next();
        if (compatibili(t,lnk))
            return true;
    }
    return false;
}
private static boolean compatibili
(TipoLinkHaLavorato t1, TipoLinkHaLavorato t2) {
    // funzione di servizio: verifica se i link t1 e t2 sono "compatibili",

```

```

// ovvero se si riferiscono alla stessa azienda e a periodi temporali
// con intersezione non nulla
return t1.getAzienda() == t2.getAzienda() && // UGUAGLIANZA SUPERFICIALE
        t2.getAnnoFine() >= t1.getAnnoInizio() &&
        t2.getAnnoInizio() <= t1.getAnnoFine();
}
private AnalisiMercatoLavoro() { }
}

```

Soluzione esercizio 13

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

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

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

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

Soluzione esercizio 13 (cont.)

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

```

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

```

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

```

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

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

```

```

        if (p.getLinkOccupazione().getDaAnno() < min)
            min = p.getLinkOccupazione().getDaAnno();
    }
    /* 2 */
    it = ins.iterator();
    boolean trovato = false;
    while (!trovato) {
        Persona p = it.next();
        if (p.getLinkOccupazione().getDaAnno() == min) {
            AssociazioneOccupazione.elimina(p.getLinkOccupazione());
            TipoLinkOccupazione t = null;
            try {
                t = new TipoLinkOccupazione(st,p,anno);
            }
            catch (EccezionePrecondizioni e) {
                System.out.println(e);
            }
            AssociazioneOccupazione.inserisci(t);
            trovato = true;
        }
    }
}
public static void libera(Set<Stanza> ins) {
    /* 1 */
    Iterator<Stanza> it = ins.iterator();
    int min = (it.next()).getLinkOccupazione().getDaAnno();

```

```

    }
    catch (EccezionePrecondizioni e) {
        System.out.println(e);
    }
    AssociazioneOccupazione.elimina(t2);
    AssociazioneOccupazione.inserisci(t1);
}
}
private RiallocazionePersonale() { }
}

```

```

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) {
        AssociazioneOccupazione.elimina(s.getLinkOccupazione());
    }
}
}
public static void trasloca
(Set<Persona> ins1, Set<Persona> ins2, int anno) {
    Iterator<Persona> it1 = ins1.iterator();
    Iterator<Persona> it2 = ins2.iterator();
    while(it1.hasNext()) {
        Persona p1 = it1.next();
        Persona p2 = it2.next();
        TipoLinkOccupazione t2 = p2.getLinkOccupazione();
        Stanza st = t2.getStanza();
        TipoLinkOccupazione t1 = null;
        try {
            t1 = new TipoLinkOccupazione(st,p1,anno);

```

Soluzione esercizio 14

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

```

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

```

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

```

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

```

Soluzione esercizio 15

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

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

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

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

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

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

240

```
        out.add(lnk.getPersona());
    }
    return out;
}
private GestioneAnagrafe() { }
}
```

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

Soluzione esercizio 16

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

```
// File MoltMax/ValutazioneDidattica.java
```

```
import java.util.*;

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

241

Soluzione esercizio 17

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

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

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

```
// File MoltMax1/CambiamentoIndirizzo.java

import java.util.*;

public final class CambiamentoIndirizzo {
    public static void cambiano(Set<Studente> i, CorsoDiLaurea c,
                                int anno)
        throws EccezioneCardMin {
        Iterator<Studente> it = i.iterator();
        while(it.hasNext()) {
            Studente s = it.next();
```

242

Soluzione esercizio 18

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

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

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

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

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

243

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

```
if (s.getLinkIscritto().getCorsoDiLaurea() != c) {
    AssociazioneIscritto.elimina(s.getLinkIscritto());
    TipoLinkIscritto t = null;
    try {
        t = new TipoLinkIscritto(c,s,anno);
    }
    catch (EccezionePrecondizioni e) {
        System.out.println(e);
    }
    AssociazioneIscritto.inserisci(t);
}
}
private CambiamentoIndirizzo() { }
}
```

Soluzione esercizio 18 (cont.)

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

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

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

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

244

Sol. eserc. 18: classe Java Facolta

```
// File MoltMax1/Esercizio/Facolta.java
import java.util.*;
public class Facolta {
    private final String nome;
    private HashSet<TipoLinkDipendente> insieme_link;
    public static final int MIN_LINK_DIPENDENTE = 8;
    public Facolta(String n) {
        nome = n;
        insieme_link = new HashSet<TipoLinkDipendente>();
    }
    public String getNome() { return nome; }
    public int quantiDipendenti() { return insieme_link.size(); }
    public void inserisciLinkDipendente(AssociazioneDipendente a) {
        if (a != null) insieme_link.add(a.getLink());
    }
    public void eliminaLinkDipendente(AssociazioneDipendente a) {
        if (a != null) insieme_link.remove(a.getLink());
    }
    public Set<TipoLinkDipendente> getLinkDipendente() throws EccezioneCardMin {
        if (quantiDipendenti() < MIN_LINK_DIPENDENTE)
            throw new EccezioneCardMin("Cardinalita' minima violata");
        else return (HashSet<TipoLinkDipendente>)insieme_link.clone();
    }
}
```

245

Sol. eserc. 18: classe Java Corso

```
// File MoltMax1/Esercizio/Corso.java
import java.util.*;
public class Corso {
    private final String nome;
    private HashSet<TipoLinkInsegna> insieme_link;
    public static final int MIN_LINK_INSEGNA = 1;
    public Corso(String n) {
        nome = n;
        insieme_link = new HashSet<TipoLinkInsegna>();
    }
    public String getNome() { return nome; }
    public int quantiInsegna() { return insieme_link.size(); }
    public void inserisciLinkInsegna(AssociazioneInsegna a) {
        if (a != null) insieme_link.add(a.getLink());
    }
    public void eliminaLinkInsegna(AssociazioneInsegna a) {
        if (a != null) insieme_link.remove(a.getLink());
    }
    public Set<TipoLinkInsegna> getLinkInsegna() throws EccezioneCardMin {
        if (quantiInsegna() < MIN_LINK_INSEGNA)
            throw new EccezioneCardMin("Cardinalita' minima violata");
        else return (HashSet<TipoLinkInsegna>)insieme_link.clone();
    }
}
```

246

Sol. eserc. 18: classe Java Docente

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

247

```

}
public void eliminaLinkDipendente(AssociazioneDipendente a) {
    if (a != null) link_dipendente = null;
}
public TipoLinkDipendente getLinkDipendente() throws EccezioneCardMin {
    if (link_dipendente == null)
        throw new EccezioneCardMin("Cardinalita' minima violata");
    else
        return link_dipendente;
}
public void inserisciLinkInsegna(AssociazioneInsegna a) {
    if (a != null) insieme_link_insegna.add(a.getLink());
}
public void eliminaLinkInsegna(AssociazioneInsegna a) {
    if (a != null) insieme_link_insegna.remove(a.getLink());
}
public Set<TipoLinkInsegna> getLinkInsegna() throws EccezioneCardMin,
                                                    EccezioneCardMax {
    if (quantiInsegna() < MIN_LINK_INSEGNA)
        throw new EccezioneCardMin("Cardinalita' minima violata");
    if (quantiInsegna() > MAX_LINK_INSEGNA)
        throw new EccezioneCardMax("Cardinalita' massima violata");
    else return (HashSet<TipoLinkInsegna>)insieme_link_insegna.clone();
}
}

```

Sol. eserc. 18: classe AssociazioneDipendente

```

// File MoltMax1/Esercizio/AssociazioneDipendente.java
public final class AssociazioneDipendente {
    private AssociazioneDipendente(TipoLinkDipendente x) { link = x; }
    private TipoLinkDipendente link;
    public TipoLinkDipendente getLink() { return link; }
    public static void inserisci(TipoLinkDipendente y) {
        if (y != null && y.getDocente().quantiDipendente() == 0) {
            AssociazioneDipendente k = new AssociazioneDipendente(y);
            k.link.getFacolta().inserisciLinkDipendente(k);
            k.link.getDocente().inserisciLinkDipendente(k);
        }
    }
    public static void elimina(TipoLinkDipendente y) {
        try {
            if (y != null && y.getDocente().getLinkDipendente().equals(y) ) {
                AssociazioneDipendente k = new AssociazioneDipendente(y);
                k.link.getFacolta().eliminaLinkDipendente(k);
                k.link.getDocente().eliminaLinkDipendente(k);
            }
        }
        catch (EccezioneCardMin e) {
            System.out.println(e);
        }
    }
}
}

```

248

Sol. eserc. 19: classe Java Studente

```

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

```

```

public Set<TipoLinkEsame> getLinkEsame() {
    return (HashSet<TipoLinkEsame>)insieme_link.clone();
}
}

```

249

Sol. eserc. 19: classe Java Professore

```
// File Ternaria/Professore.java
import java.util.*;
public class Professore {
    private final String codice;
    private int eta;
    private HashSet<TipoLinkEsame> insieme_link;
    public Professore(String c, int e) {
        codice = c;
        eta = e;
        insieme_link = new HashSet<TipoLinkEsame>();
    }
    public String getCodice() { return codice; }
    public int getEta() { return eta; }
    public void setEta(int e) { eta = e; }
    public void inserisciLinkEsame(AssociazioneEsame a) {
        if (a != null) insieme_link.add(a.getLink());
    }
    public void eliminaLinkEsame(AssociazioneEsame a) {
        if (a != null) insieme_link.remove(a.getLink());
    }
    public Set<TipoLinkEsame> getLinkEsame() {
        return (HashSet<TipoLinkEsame>)insieme_link.clone();
    }
}
```

250

Sol. eserc. 19: classe Java Corso

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

251

Sol. eserc. 19: classe Java AssociazioneEsame

```
// File Ternaria/AssociazioneEsame.java
public final class AssociazioneEsame {
    private AssociazioneEsame(TipoLinkEsame x) { link = x; }
    private TipoLinkEsame link;
    public TipoLinkEsame getLink() { return link; }
    public static void inserisci(TipoLinkEsame y) {
        if (y != null) {
            AssociazioneEsame k = new AssociazioneEsame(y);
            k.link.getStudiante().inserisciLinkEsame(k);
            k.link.getProfessore().inserisciLinkEsame(k);
        }
    }
    public static void elimina(TipoLinkEsame y) {
        if (y != null) {
            AssociazioneEsame k = new AssociazioneEsame(y);
            k.link.getStudiante().eliminaLinkEsame(k);
            k.link.getProfessore().eliminaLinkEsame(k);
        }
    }
}
```

252

Sol. eserc. 19: classe Java TipoLinkEsame

```
// File Ternaria/TipoLinkEsame.java
public class TipoLinkEsame {
    private final Corso ilCorso;
    private final Professore ilProfessore;
    private final Studente loStudiante;
    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; loStudiante = y; ilProfessore = z; voto = a;
    }
    public boolean equals(Object o) {
        if (o != null && getClass().equals(o.getClass())) {
            TipoLinkEsame b = (TipoLinkEsame)o;
            return b.ilCorso == ilCorso && b.loStudiante == loStudiante &&
                b.ilProfessore == ilProfessore;
        }
        else return false;
    }
    public int hashCode() {
        return ilCorso.hashCode() + loStudiante.hashCode() +
            ilProfessore.hashCode();
    }
}
```

253

Sol. eserc. 19: use case

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

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

254

Sol. eserc. 20: classe Java Azienda

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

public class Azienda {
    private final String nome;
    private final int annoFondazione;
    private HashSet<TipoLinkContenzioso> linkAccusatrice;
    private HashSet<TipoLinkContenzioso> linkAccusata;
    public Azienda(String no, int an) {
        nome = no;
        annoFondazione = an;
        linkAccusatrice = new HashSet<TipoLinkContenzioso>();
        linkAccusata = new HashSet<TipoLinkContenzioso>();
    }
    public String getNome() { return nome; }
    public int getAnnoFondazione() { return annoFondazione; }
    public Set<TipoLinkContenzioso> getAccusatrice() {
        return (HashSet<TipoLinkContenzioso>)linkAccusatrice.clone();
    }
    public Set<TipoLinkContenzioso> getAccusata() {
        return (HashSet<TipoLinkContenzioso>)linkAccusata.clone();
    }
    public void inserisciLinkAccusatrice(AssociazioneContenzioso a) {
        if (a != null) { linkAccusatrice.add(a.getLink()); }
    }
```

```
    public void eliminaLinkAccusatrice(AssociazioneContenzioso a) {
        if (a != null) { linkAccusatrice.remove(a.getLink()); }
    }
    public void inserisciLinkAccusata(AssociazioneContenzioso a) {
        if (a != null) { linkAccusata.add(a.getLink()); }
    }
    public void eliminaLinkAccusata(AssociazioneContenzioso a) {
        if (a != null) { linkAccusata.remove(a.getLink()); }
    }
    public String toString() {
        return nome + " " + annoFondazione;
    }
}
```

255

Sol. eserc. 20: classe Java Persona

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

public class Persona {
    private final String nome, cognome, codiceFiscale;
    private HashSet<TipoLinkContenzioso> linkContenzioso;
    public Persona(String no, String co, String cf) {
        nome = no;
        cognome = co;
        codiceFiscale = cf;
        linkContenzioso = new HashSet<TipoLinkContenzioso>();
    }
    public String getNome() { return nome; }
    public String getCognome() { return cognome; }
    public String getCodiceFiscale() { return codiceFiscale; }
    public Set<TipoLinkContenzioso> getContenzioso() {
        return (HashSet<TipoLinkContenzioso>)linkContenzioso.clone();
    }
    public void inserisciLinkContenzioso(AssociazioneContenzioso a) {
        if (a != null) { linkContenzioso.add(a.getLink()); }
    }
    public void eliminaLinkContenzioso(AssociazioneContenzioso a) {
        if (a != null) { linkContenzioso.remove(a.getLink()); }
    }
}
```

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

256

Sol. eserc. 20: classe Java TipoLinkContenzioso

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

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

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

257

Sol. eserc. 20: AssociazioneContenzioso

```
// File Ternaria/Esercizio/AssociazioneContenzioso.java

public final class AssociazioneContenzioso {
    private AssociazioneContenzioso(TipoLinkContenzioso x) { link = x; }
    private TipoLinkContenzioso link;
    public TipoLinkContenzioso getLink() { return link; }
    public static void inserisci(TipoLinkContenzioso y) {
        if (y != null) {
            AssociazioneContenzioso k = new AssociazioneContenzioso(y);
            y.getAccusatrice().inserisciLinkAccusatrice(k);
            y.getAccusata().inserisciLinkAccusata(k);
            y.getGiudice().inserisciLinkContenzioso(k);
        }
    }
    public static void elimina(TipoLinkContenzioso y) {
        if (y != null) {
            AssociazioneContenzioso k = new AssociazioneContenzioso(y);
            y.getAccusatrice().eliminaLinkAccusatrice(k);
            y.getAccusata().eliminaLinkAccusata(k);
            y.getGiudice().eliminaLinkContenzioso(k);
        }
    }
}
```

258

Sol. eserc. 21: Specifica dello use case

InizioSpecificaUseCase ControlloContenziosi

DueContenziosi (S : *Insieme(Azienda)*): *Insieme(Azienda)*

pre: true // nessuna

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

Accusata (A : *Azienda*, P : *Persona*): *booleano*

pre: true // nessuna

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

FineSpecifica

259

Sol. eserc. 21: Algor. per le oper. dello use-case

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

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

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

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

260

Sol. eserc. 21: Realizzazione dello use case

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

```
import java.util.*;

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

261

Soluzione esercizio 22

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

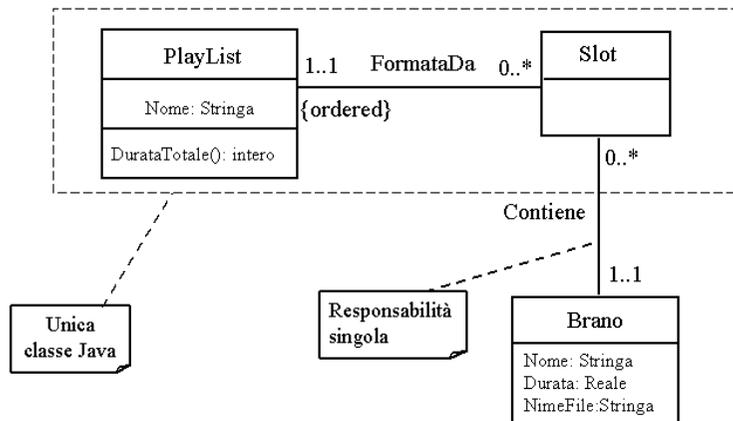
In particolare:

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

262

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

Soluzione esercizio 22 (cont.)



263

Sol. exerc. 22: classe Java PlayList

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

264

```

public void eliminaOgniOccorrenzaLinkContiene(Brano b) {
    if (b != null) {
        while(sequenza_link.contains(b))
            sequenza_link.remove(b);
    }
}
public List<Brano> getLinkContiene() {
    return (LinkedList<Brano>)sequenza_link.clone();
}
public int durataTotale() {
    int result = 0;
    Iterator<Brano> ib = sequenza_link.iterator();
    while (ib.hasNext()) {
        Brano b = ib.next();
        result = result + b.getDurata();
    }
    return result;
}
}
}

```

Soluzione esercizio 23

Si lascia il dettaglio della fase di progetto per esercizio.

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

La struttura dei file e dei package è la seguente:

```

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

```

265

Sol. exerc. 23: classe Java Universita

```

// File PackageUniversita/Universita.java

package PackageUniversita;
import java.util.*;

public class Universita {
    private final String nome;
    private HashSet<TipoLinkIscrizione> insieme_link;
    public Universita(String n) {
        nome = n;
        insieme_link = new HashSet<TipoLinkIscrizione>();
    }
    public String getNome() { return nome; }
    public void inserisciLinkIscrizione(AssociazioneIscrizione a) {
        if (a != null) insieme_link.add(a.getLink());
    }
    public void eliminaLinkIscrizione(AssociazioneIscrizione a) {
        if (a != null) insieme_link.remove(a.getLink());
    }
    public Set<TipoLinkIscrizione> getLinkIscrizione() {
        return (HashSet<TipoLinkIscrizione>)insieme_link.clone();
    }
}

```

266

Sol. exerc. 23: classe Java Citta

```

// File PackageUniversita/Citta.java

package PackageUniversita;

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

```

267

Sol. eserc. 23: classe Java Persona

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

```
package PackageUniversita.Persona;
import PackageUniversita.*;

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

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

268

Sol. eserc. 23: classe Java Studente

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

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

public class Studente extends Persona {
    protected final String matricola;
    protected TipoLinkIscrizione iscrizione;
    public static final int MIN_LINK_ISCRIZIONE = 1;
    public Studente(String n, String c, String m) {
        super(n,m);
        matricola = m;
    }
    public String getMatricola() { return matricola; }
    public int quantiIscrizione() {
        if (iscrizione == null)
            return 0;
        else return 1;
    }
    public TipoLinkIscrizione getLinkIscrizione() throws EccezioneCardMin {
        if (iscrizione == null)
            throw new EccezioneCardMin("Cardinalita' minima violata");
        else
```

```
            return iscrizione;
        }
    public void inserisciLinkIscrizione(AssociazioneIscrizione a) {
        if (a != null) iscrizione = a.getLink();
    }
    public void eliminaLinkIscrizione(AssociazioneIscrizione a) {
        if (a != null) iscrizione = null;
    }
}
```

269

Sol. eserc. 23: classe Java AssociazioneIscrizione

```
// File PackageUniversita/AssociazioneIscrizione.java
package PackageUniversita;
import PackageUniversita.Studente.*;

public final class AssociazioneIscrizione {
    private AssociazioneIscrizione(TipoLinkIscrizione x) { link = x; }
    private TipoLinkIscrizione link;
    public TipoLinkIscrizione getLink() { return link; }
    public static void inserisci(TipoLinkIscrizione y) {
        if (y != null && y.getStudiante().quantiIscrizione() == 0) {
            AssociazioneIscrizione k = new AssociazioneIscrizione(y);
            k.link.getUniversita().inserisciLinkIscrizione(k);
            k.link.getStudiante().inserisciLinkIscrizione(k);
        }
    }
    public static void elimina(TipoLinkIscrizione y) {
        try {
            if (y != null && y.getStudiante().getLinkIscrizione().equals(y) ) {
                AssociazioneIscrizione k = new AssociazioneIscrizione(y);
                k.link.getUniversita().eliminaLinkIscrizione(k);
                k.link.getStudiante().eliminaLinkIscrizione(k);
            }
        }
        catch (EccezioneCardMin e) {
            System.out.println(e);
        }
    }
}
```

270

Sol. eserc. 23: classe Java StatisticheIscrizioni

```
// File PackageUniversita/StatisticheIscrizioni.java
package PackageUniversita;
import PackageUniversita.Studente.*;
import java.util.*;

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

271