

Appunti di Progettazione del Software

Canale 1 · Prof. Paolo Liberatore

Capitolo 7: Dalla specifica alle classi: le visite ospedaliere

Autore: Tommaso C.

Appunti tratti dalle lezioni in aula e dal materiale del corso

Argomenti: La specifica · Schema UML · Gerarchia completa · Molteplicità e
navigabilità
Specifiche esplicite e assunzioni · Classi Java · Associazioni nei due versi · Programma di
prova

Indice

1	Dalla specifica alle classi	1
2	Lo schema delle classi	1
3	La gerarchia completa	3
4	In che verso si percorre un'associazione	4
5	Specifiche esplicite e assunzioni di dominio	5
6	Le classi Java	6
7	Il programma di prova	11
8	Esercizi	12
9	Soluzioni	14

1 Dalla specifica alle classi

Proposizione – Dalla specifica alle classi

1. Dalla specifica si disegna lo **schema UML dei dati**: classi, attributi, generalizzazioni, associazioni e molteplicità.
2. Accanto allo schema si elenca ciò che lo schema non dice: in che verso si percorre ogni associazione, quali dati sono noti alla creazione, quali si possono modificare.
3. Le classi Java si scrivono guardando solo lo schema e l'elenco, senza tornare alla specifica.
4. Un **programma di prova** crea oggetti, li collega e li stampa.

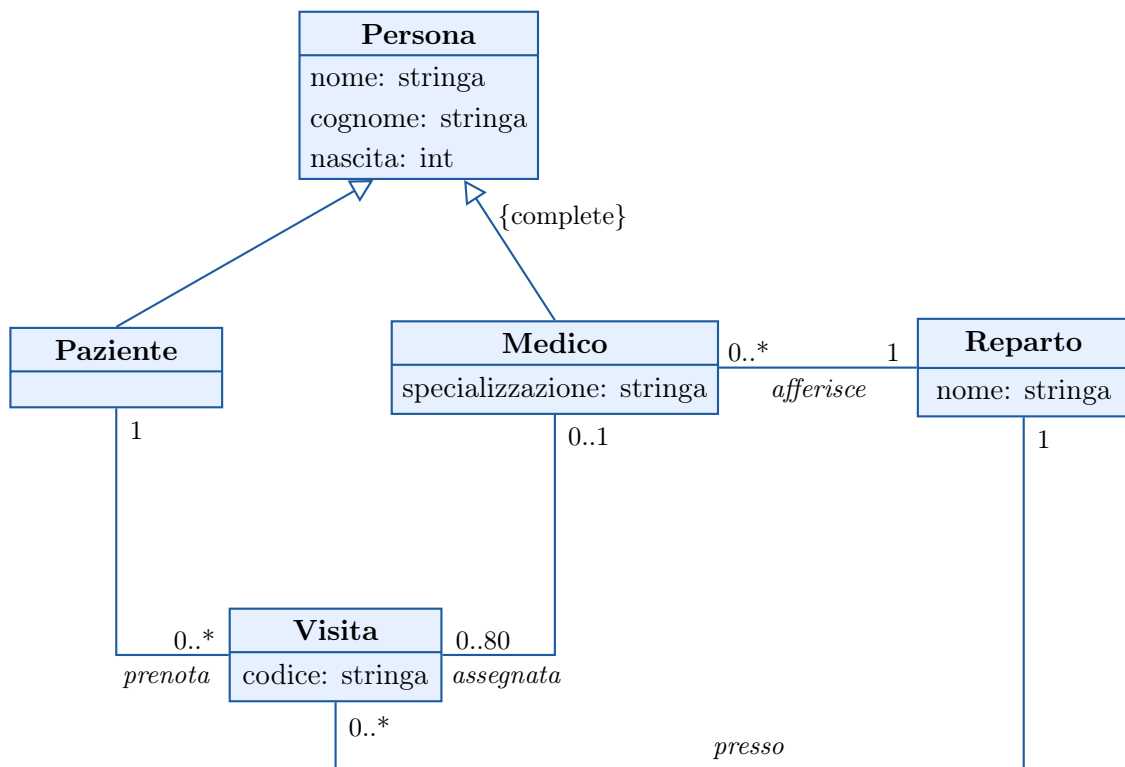
Il caso di studio del capitolo è un piccolo gestionale ospedaliero.

Nota – La specifica

L'applicazione gestisce le visite mediche di un reparto ospedaliero: pazienti, medici, visite e reparti. Di tutte le persone, quindi sia pazienti che medici, interessano nome, cognome e anno di nascita. Dei medici interessa la specializzazione (una stringa). I pazienti possono prenotare una visita presso un reparto. In seguito, la visita può venire assegnata a un medico. Ogni visita ha un codice (una stringa). Ogni paziente può prenotare quante visite vuole, ma ogni visita viene prenotata da un solo paziente. La visita è sempre e soltanto in un reparto. A un medico possono venire assegnate al massimo ottanta visite, ma non c'è un minimo. Quando la visita viene prenotata non è assegnata a nessun medico, e nel seguito può venire assegnata a un singolo medico, ma non di più. Un metodo da non implementare permette di disassegnare tutte le visite a un medico dato. Un medico afferisce a uno e un solo reparto.

2 Lo schema delle classi

Le classi si leggono nella prima frase: **Paziente**, **Medico**, **Visita**, **Reparto**. In più, “di tutte le persone” interessano nome, cognome e anno di nascita: pazienti e medici sono persone, quindi c'è una superclasse **Persona**.



Osservazione – Rettangoli di classi, non di oggetti

Le figure della memoria del Capitolo 2 sono simili, ma lì ogni rettangolo è un oggetto. Qui ogni rettangolo è una classe, cioè un insieme di oggetti.

Che cosa sta nei rettangoli.

- Nome, cognome e nascita stanno solo in **Persona**: medici e pazienti li ereditano (Capitolo 3), e ripeterli sarebbe ridondante.
- **Medico** aggiunge la specializzazione. **Paziente** non aggiunge attributi: si distingue dal medico perché prenota visite, cioè per un'associazione.
- I nomi delle classi sono al singolare e con l'iniziale maiuscola: la classe dei pazienti è **Paziente**. Java accetta anche la minuscola, ma la convenzione è questa.

Nota – Dalla frase alla molteplicità

Frase della specifica	Molteplicità	Vicino a
ogni visita viene prenotata da un solo paziente	1	Paziente
ogni paziente prenota quante visite vuole	0..*	Visita
in seguito la visita va a un singolo medico	0..1	Medico
a un medico al massimo 80 visite, nessun minimo	0..80	Visita
la visita è sempre e soltanto in un reparto	1	Reparto
un medico afferisce a uno e un solo reparto	1	Reparto

Dove la specifica non dice nulla (quante visite ha un reparto, quanti medici vi afferiscono) vale 0..*.

La molteplicità si legge dall'altro estremo: “una visita è prenotata da 1 paziente”, “una visita è assegnata a 0..1 medici”. Il verso non è intuitivo, e scambiare i due estremi è un errore frequente.

1 oppure 0..1. “Sempre” dà 1: una visita non esiste mai senza paziente o senza reparto. “In seguito” dà 0..1: una visita appena prenotata non ha medico, e lo riceve dopo. La visita ha quindi due situazioni, con o senza medico, e il campo che la rappresenta deve ammetterle entrambe.

3 La gerarchia completa

Definizione – Generalizzazione completa

Una generalizzazione è **completa** se ogni istanza della superclasse è istanza di almeno una sottoclasse: le sottoclassi coprono tutta la superclasse. Nel diagramma si scrive {complete} accanto alle frecce.

Qui la specifica dice “di tutte le persone, quindi sia pazienti che medici”: le uniche persone che interessano sono pazienti e medici, e la generalizzazione è completa.

Proposizione – Completa in Java

In Java una generalizzazione completa si realizza dichiarando la superclasse **abstract**: non esistono persone che non siano pazienti o medici.

Le due regole insieme danno una partizione. Le sottoclassi Java sono sempre disgiunte (Capitolo 3); con **abstract** la cornice di **Persona** è vuota (Capitolo 5). Quindi ogni persona è un paziente oppure un medico, mai tutti e due e mai nessuno dei due. **new**

`Persona(...)` non compila (*Persona is abstract; cannot be instantiated*), mentre `Persona x = new Medico(...)` va bene.

Osservazione – Un medico che si fa visitare

Un medico può farsi visitare nel proprio ospedale e quindi essere anche un paziente. Con queste classi non c'è un oggetto che sia insieme **Medico** e **Paziente**: servono due oggetti, con gli stessi nome, cognome e anno di nascita.

Se interessassero anche infermieri o impiegati, sarebbero altre sottoclassi di **Persona**: la scelta di **abstract** si rifà quando cambiano le persone di interesse (Capitolo 5, “Astratta o no: lo dicono i requisiti”).

4 In che verso si percorre un'associazione

Definizione – Navigabilità

Un'associazione fra C e D è **navigabile** da C a D se, dato un oggetto di C , il programma deve poter trovare gli oggetti di D collegati. In Java ogni verso navigabile è un campo di C : un riferimento se il massimo dal lato di D è 1, un insieme altrimenti.

Lo schema non dice in che verso si percorre un'associazione. La scelta dipende da:

1. **requisiti espliciti**: una funzionalità richiesta che deve partire da un oggetto e arrivare a quelli collegati;
2. **uso normale del dominio**: domande che ci si aspetta di fare, anche se la specifica non le scrive;
3. **costo**: ogni associazione navigabile nei due versi è memorizzata due volte, e i due collegamenti vanno tenuti coerenti (Capitolo 2).

Esempio – Disassegnare le visite di un medico

La specifica chiede che si possa scrivere un metodo che disassegna tutte le visite di un medico dato. Per disassegnarle bisogna sapere quali sono: dato il medico, si deve arrivare alle sue visite. La frase chiede solo una funzionalità, ma implica che *assegnata* sia navigabile da **Medico** a **Visita**, e quindi che **Medico** abbia l'insieme delle sue visite. Esercizio 2.

Nel caso di studio:

Associazione	Versi	Motivo
prenota	tutti e due	dominio: le visite di un paziente, il paziente di una visita
assegnata	tutti e due	requisito: disassegnare; dominio: il medico della visita
presso	da Visita	dominio: il reparto della visita
afferisce	da Medico	dominio: il reparto del medico

Nessun requisito chiede a un reparto le sue visite o i suoi medici: per presso e afferisce basta un verso, e non c'è coerenza da mantenere.

5 Specifiche esplicite e assunzioni di dominio

Definizione – Assunzione di dominio

Un'**assunzione di dominio** è una proprietà che la specifica non scrive e che chi la scrive dà per scontata. Una specifica **esplicita** elenca, per ogni classe, anche quali dati e collegamenti sono noti alla creazione e quali si possono modificare.

Queste informazioni decidono costruttori e setter: un dato noto alla creazione è un argomento del costruttore, un dato modificabile ha un setter. Compilando l'elenco ci si accorge di che cosa la specifica non ha detto.

Nota – L'elenco del caso di studio

Dato o collegamento	Alla creazione	Dopo
nome, cognome, nascita	noti	non cambiano
specializzazione di un medico	nota	non cambia
visite prenotate da un paziente	nessuna	se ne aggiungono
codice e paziente di una visita	noti	non cambiano
medico di una visita	nessuno	si assegna, si toglie, cambia
visite assegnate a un medico	nessuna	se ne aggiungono e tolgono

Che cosa la specifica non dice. La tabella contiene tre assunzioni:

- che nome, cognome e nascita non cambino: è una scelta, perché nella realtà nome e cognome possono cambiare;
- che la specializzazione di un medico non cambi;
- che il paziente esista prima della visita e che la visita non passi a un altro paziente: le visite sono personali.

Sono ovvie per chi conosce gli ospedali. In un dominio specialistico le stesse lacune lasciano il programmatore senza risposta.

Osservazione – Ciò che sembra ovvio può essere falso

Che un codice fiscale sia di 16 caratteri fra lettere e cifre vale per le persone fisiche; quello di una società è di 11 cifre e coincide con la partita IVA. Una classe che controlla solo la prima forma rifiuta codici validi.

6 Le classi Java

Nella prima versione i collegamenti nei due versi si aggiornano a mano. Le classi applicano le regole del Capitolo 4: campi **private**, getter per tutti, setter solo per ciò che si può modificare, controlli nel costruttore.

```
public abstract class Persona {
    private String nome;
    private String cognome;
    private int nascita;    // anno di nascita

    Persona(String nome, String cognome, int nascita) {
        if (nome == null || nome.isEmpty()
            || cognome == null || cognome.isEmpty()) {
            throw new RuntimeException("Nome o cognome vuoto");
        }
        this.nome = nome;
        this.cognome = cognome;
        this.nascita = nascita;
    }

    String getNome() {
        return nome;
    }

    String getCognome() {
        return cognome;
    }

    int getNascita() {
        return nascita;
    }
}
```

Senza setter, i tre dati si impostano solo nel costruttore e non cambiano più. Il controllo sta in **Persona** e non in **Paziente**: ogni sottoclasse chiama questo costruttore con **super**, e il controllo vale per pazienti e medici.

```
import java.util.HashSet;

public class Paziente extends Persona {
    private HashSet<Visita> prenota;    // visite prenotate

    Paziente(String nome, String cognome, int nascita) {
        super(nome, cognome, nascita);
        this.prenota = new HashSet<Visita>();
    }

    HashSet<Visita> getPrenota() {
        return prenota;
    }

    // toString più avanti
}
```

Paziente non ha attributi propri, ma ha un costruttore: **Persona** ha solo un costruttore con argomenti (Capitolo 3). L'insieme delle visite non è un argomento: un paziente nasce senza visite, e l'insieme si crea vuoto.

```
import java.util.HashSet;

public class Medico extends Persona {
    private String specializzazione;
    private HashSet<Visita> assegnata;    // visite assegnate

    Medico(String nome, String cognome, int nascita,
            String specializzazione) {
        super(nome, cognome, nascita);
        this.specializzazione = specializzazione;
        this.assegnata = new HashSet<Visita>();
    }

    String getSpecializzazione() {
        return specializzazione;
    }

    HashSet<Visita> getAssegnata() {
        return assegnata;
    }

    public String toString() {
        return "dott. " + getNome() + " " + getCognome() + " ("
            + getNascita() + "), " + specializzazione;
    }
}
```

Il costruttore riceve quattro argomenti, nome, cognome, nascita e specializzazione, e passa i primi tre a **super**. Se il costruttore di **Persona** manca, Java usa quello di default senza argomenti: **super()** compila, ma nome, cognome e nascita non sono mai impostati. Si scrive quindi prima il costruttore della superclasse, poi quelli delle sottoclassi.

Il toString di una sottoclasse. Deve stampare tutti i dati, compresi quelli ereditati. Questi sono **private** in **Persona** e la sottoclasse non può nominarli: li legge con i getter. Una stampa sintetica (**dott. Laura Verdi (1975), cardiologia**) si legge meglio di una che ripete **nome=, cognome=**.

```
public class Visita {
    private String codice;
    private Paziente prenota;    // chi l'ha prenotata
    private Medico assegnata;    // null finché non è assegnata

    Visita(String codice, Paziente prenota) {
        if (prenota == null) {
            throw new RuntimeException("Visita senza paziente");
        }
        this.codice = codice;
        this.prenota = prenota;
    }

    String getCodice() {
        return codice;
    }

    Paziente getPrenota() {
        return prenota;
    }

    Medico getAssegnata() {
        return assegnata;
    }

    void setAssegnata(Medico assegnata) {
        this.assegnata = assegnata;
    }

    // toString più avanti
}
```

Ogni scelta viene dall'elenco della Sezione 5:

- codice e paziente sono noti alla creazione e non cambiano: argomenti del costruttore, solo getter;
- il paziente ha molteplicità 1: il costruttore rifiuta `null`;
- il medico non è noto alla creazione e cambia: niente costruttore, getter e setter, e `null` rappresenta la visita non assegnata.

Osservazione – Getter e setter per convenzione

Il medico di una visita può diventare `null` e cambiare liberamente: un campo accessibile non permetterebbe nessun uso sbagliato. Lo si rende comunque **private**, con getter e setter, perché chi usa le classi si aspetta di trovare **getAssegnata** e **setAssegnata** e di non accedere ai campi. Non è una questione di correttezza, ma di convenzione.

Stampare un'associazione nei due versi. Il paziente conosce le sue visite e ogni visita il suo paziente. Se il `toString` di **Paziente** stampa le visite e quello di **Visita** stampa il paziente, la stampa non termina: il paziente stampa la visita, che stampa il paziente, che stampa di nuovo la visita. Il programma si ferma con *java.lang.StackOverflowError*.

Proposizione – toString e associazioni nei due versi

In un'associazione navigabile nei due versi, al più uno dei due `toString` stampa per intero l'oggetto collegato; l'altro ne stampa solo qualche dato.

```
// in Paziente
public String toString() {
    return getNome() + " " + getCognome() + " (" + getNascita()
        + "), visite " + prenota;
}

// in Visita
public String toString() {
    String s = codice + " di " + prenota.getNome() + " "
        + prenota.getCognome();
    if (assegnata != null) {
        s = s + ", dott. " + assegnata.getCognome();
    }
    return s;
}
```

7 Il programma di prova

```
public class Prova {
    public static void main(String[] args) {
        Paziente p = new Paziente("Mario", "Rossi", 1980);
        Visita v = new Visita("V001", p);
        p.getPrenota().add(v);           // l'altro verso di prenota
        System.out.println(p);

        Medico m = new Medico("Laura", "Verdi", 1975, "cardiologia");
        System.out.println(m);
        v.setAssegnata(m);
        m.getAssegnata().add(v);         // l'altro verso di assegnata
        System.out.println(v);
        System.out.println(m.getAssegnata().size());
    }
}
```

Stampa:

```
Mario Rossi (1980), visite [V001 di Mario Rossi]
dott. Laura Verdi (1975), cardiologia
V001 di Mario Rossi, dott. Verdi
1
```

Creare la visita non basta perché compaia fra quelle del paziente: il costruttore registra il paziente nella visita, ma l'altro verso va aggiunto con `p.getPrenota().add(v)`. Lo stesso per il medico: `setAssegnata` e l'aggiunta all'insieme del medico sono due istruzioni.

Esempio – Che cosa può andare storto

Con questa versione nulla impedisce due errori di programmazione:

```
Paziente q = new Paziente("Anna", "Bianchi", 1992);
q.getPrenota().add(v); // v è di Mario Rossi
Medico m2 = new Medico("Luca", "Neri", 1980, "ortopedia");
v.setAssegnata(m2);    // m2 non ha v fra le sue visite
```

1. Un paziente ha fra le sue visite una visita prenotata da un altro paziente.
2. Una visita è assegnata a un medico ma non è fra le visite di quel medico, e resta fra quelle del medico precedente.

Sono le inconsistenze del Capitolo 2. Anche il vincolo `0..80` non è garantito: il getter restituisce l'insieme, e chiunque vi aggiunge un'ottantunesima visita (Capitolo 4, "Campi che sono collezioni"). Una versione che controlla i collegamenti richiede gli strumenti del Capitolo 4 per gli insiemi.

Osservazione – Il programma di prova si conserva

Il programma di prova non si butta quando le classi funzionano: quando emerge un errore permette di ricreare subito i dati e provare le classi da sole, senza l'interfaccia dell'applicazione. Per lo stesso motivo conviene stampare gli oggetti prima e dopo ogni operazione.

Riepilogo

- **Metodo:** specifica, schema UML, elenco di ciò che lo schema non dice, classi Java dal solo schema, programma di prova.
- **Molteplicità:** si legge dall'altro estremo; “sempre” dà **1**, “in seguito” dà **0..1**; se la specifica tace, **0..***.
- **Completa:** **{complete}**, ogni persona è paziente o medico; in Java superclasse **abstract**, e con le classi disgiunte è una partizione.
- **Navigabilità:** un verso navigabile è un campo; la decidono i requisiti espliciti, l'uso del dominio e il costo della coerenza.
- **Assunzioni:** per ogni dato, se è noto alla creazione (costruttore) e se cambia (setter); ciò che sembra ovvio va scritto.
- **Costruttori:** prima quello della superclasse; le sottoclassi chiamano **super**, e i controlli in **Persona** valgono per tutte.
- **Due versi:** collegamenti aggiornati a mano, a rischio di inconsistenze; un solo **toString** stampa per intero l'oggetto collegato.
- **Prova:** un **main** che crea, collega e stampa; si conserva.

8 Esercizi

Esercizio 1 – Il reparto

Aggiungere alle classi della Sezione 6 la classe **Reparto**, con il nome, e le associazioni *presso* e *afferisce* dello schema, navigabili come nella tabella della Sezione 4. Un reparto non cambia nome.

Soluzione a pagina 14 →

Esercizio 2 – Disassegnare le visite di un medico

Con le classi della Sezione 6, scrivere un metodo `static void disassegna(Medico m)` che disassegna tutte le visite del medico `m`, lasciando coerenti i due versi di *assegnata*.

[Soluzione a pagina 15 →](#)

Esercizio 3 – Appello del 19/02/2021, Canale 2, Prof. Giuseppe De Giacomo, domanda 1, adattato

L'applicazione riguarda un videogioco sul mondo del crimine. Una gilda ha un nome ed è composta da un certo numero di assassini, minimo 3. Un assassino è una persona (vedi dopo) e appartiene a esattamente una gilda. A un assassino può essere commissionato un tentativo di omicidio con un codice (una stringa), una persona che è la vittima ed un'altra persona che è il committente. Un omicidio ha due campi booleani per memorizzare rispettivamente se concluso o meno e in caso sia concluso se è riuscito o fallito. Se è concluso e riuscito allora è di interesse memorizzare il tempo impiegato per eseguirlo, in secondi. Tutte le persone hanno nome, cognome e indirizzo, e tra queste gli assassini hanno anche un'arma preferita (una stringa), e gli assassini esperti hanno anche una seconda arma.

Basandosi sui requisiti riportati sopra, produrre il diagramma delle classi in UML per l'applicazione.

[Soluzione a pagina 16 →](#)

9 Soluzioni

Soluzione dell'Esercizio 1 – Il reparto

Il reparto ha solo il nome, noto alla creazione e non modificabile:

```
public class Reparto {
    private String nome;

    Reparto(String nome) {
        this.nome = nome;
    }

    String getNome() {
        return nome;
    }

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

Le due associazioni sono navigabili solo da **Visita** e da **Medico**: il campo sta lì, e **Reparto** non ha insiemi. Dal lato di **Reparto** la molteplicità è **1** in entrambe: il reparto è noto alla creazione, quindi va nel costruttore, che rifiuta **null**.

```
// in Visita
private Reparto presso;

Visita(String codice, Paziente prenota, Reparto presso) {
    if (prenota == null || presso == null) {
        throw new RuntimeException("Visita senza paziente o reparto");
    }
    this.codice = codice;
    this.prenota = prenota;
    this.presso = presso;
}

Reparto getPresso() {
    return presso;
}
```

```
// in Medico
private Reparto afferisce;

Medico(String nome, String cognome, int nascita,
```

```

        String specializzazione, Reparto afferisce) {
    super(nome, cognome, nascita);
    if (afferisce == null) {
        throw new RuntimeException("Medico senza reparto");
    }
    this.specializzazione = specializzazione;
    this.afferisce = afferisce;
    this.assegnata = new HashSet<Visita>();
}

Reparto getAfferisce() {
    return afferisce;
}

```

Con un solo verso non c'è un secondo collegamento da aggiornare: creare la visita basta. Se un medico può cambiare reparto (la specifica non lo dice) si aggiunge un setter che rifiuta `null`.

[← Torna all'Esercizio 1, pagina 12](#)

Soluzione dell'Esercizio 2 – Disassegnare le visite di un medico

Grazie all'insieme delle visite del medico, si scorrono solo le sue visite: a ciascuna si toglie il medico, poi si svuota l'insieme.

```

static void disassegna(Medico m) {
    for (Visita v : m.getAssegnata()) {
        v.setAssegnata(null);
    }
    m.getAssegnata().clear();
}

```

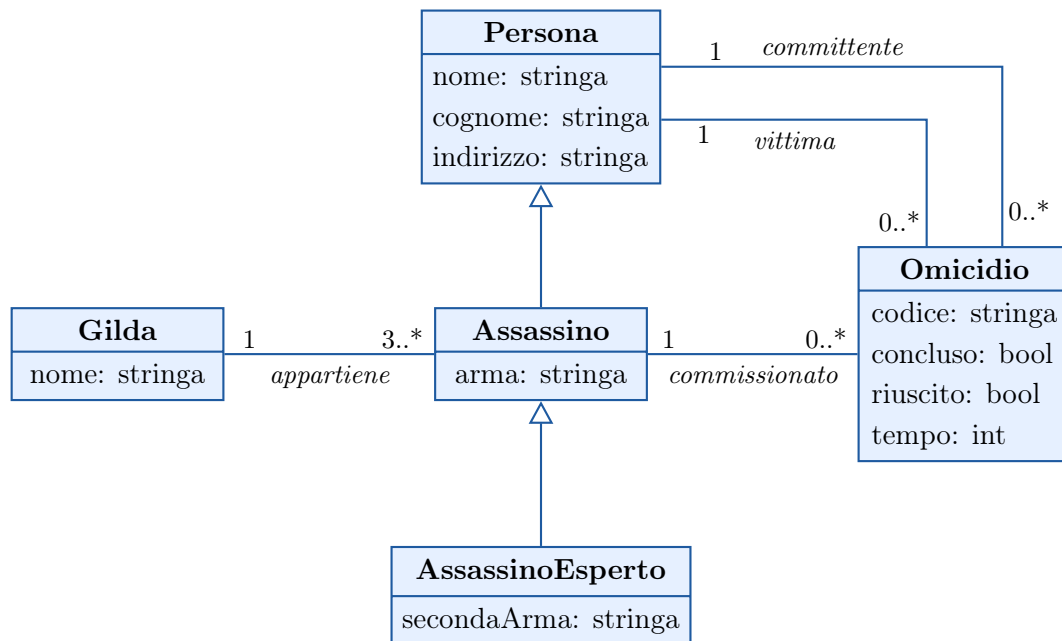
Senza quell'insieme si dovrebbero scorrere tutte le visite dell'applicazione, cercando quelle con `getAssegnata() == m`. Il metodo `clear()` toglie tutti gli elementi di una collezione.

Errore tipico. Togliere ogni visita dentro il ciclo, con `m.getAssegnata().remove(v)`, modifica l'insieme mentre il `foreach` lo scorre: con due o più visite il programma si ferma con `java.util.ConcurrentModificationException`.

[← Torna all'Esercizio 2, pagina 13](#)

Soluzione dell'Esercizio 3 – Appello del 19/02/2021, Canale 2, Prof. Giuseppe De Giacomo, domanda 1, adattato

Le classi sono **Gilda**, **Persona**, **Assassino**, **AssassinoEsperto** e **Omicidio**. Gli assassini sono persone e gli assassini esperti sono assassini: una gerarchia a due livelli. Non è completa, perché la specifica parla di vittime e committenti che sono persone qualsiasi.



- *appartiene*: ogni assassino ha esattamente una gilda (1 vicino a **Gilda**); una gilda ha almeno tre assassini (3..* vicino a **Assassino**).
- *commissionato*: ogni omicidio è commissionato a un assassino (1); a un assassino se ne commissionano quanti si vuole (0..*).
- *vittima* e *committente* sono due associazioni distinte fra le stesse classi: ogni omicidio ha una vittima e un committente (1), e una persona può essere vittima o committente di più omicidi. Vittima e committente possono essere anche assassini, perché gli assassini sono persone.

Il codice, i due booleani e il tempo (in secondi, **int**) sono attributi di **Omicidio**; le due armi stanno in **Assassino** e in **AssassinoEsperto**, che eredita la prima. Tre frasi non entrano nello schema e vanno nell'elenco accanto: il tempo ha senso solo se l'omicidio è concluso e riuscito, **riuscito** solo se è concluso, e il committente è un'altra persona rispetto alla vittima.

[← Torna all'Esercizio 3, pagina 13](#)