20 Basi di dati a oggetti e multimediali

Le basi di dati a oggetti, sviluppate a partire dalla seconda metà degli anni Ottanta, integrano la tecnologia delle basi di dati con il paradigma a oggetti, sviluppato nell'ambito dei linguaggi di programmazione e utilizzato, sul piano metodologico, nell'ambito dell'ingegneria del software. Nelle basi di dati a oggetti ogni entità del mondo reale è rappresentata da un *oggetto*. Esempi di oggetti possono essere:

- componenti elettronici, progettati tramite un sistema di *Computer-Aided Design* (CAD);
- componenti meccanici, progettati tramite un sistema di *Computer-Aided Manufacturing* (CAM);
- specifiche e programmi, gestiti da un ambiente di *Computer-Aided Software Engineering* (CASE).

Vi sono poi altre tipologie di oggetti che richiedono un trattamento particolare, più ricco di quello che i normali sistemi relazionali riescono normalmente a offrire, che rientrano nell'area delle basi di dati multimediali:

- dati multimediali, che comprendono testi liberi, immagini e suoni, collezionati tramite *sistemi per la gestione di documenti multimediali*;
- dati spaziali o geografici, che descrivono per esempio figure geometriche o mappe geografiche, gestiti dai *sistemi informativi geografici* (GIS).

Queste applicazioni presentano caratteristiche abbastanza differenti fra loro. Tuttavia è comune a tutti questi campi applicativi il bisogno di organizzare i dati e gestirli in un modo diverso dai sistemi relazionali. Per quanto riguarda i sistemi a oggetti, nel modello relazionale ogni oggetto si trova distribuito su di un alto numero di tabelle; una visione unitaria dell'oggetto richiede query complesse che lo ricostruiscano estraendone i componenti dalle varie tabelle, tramite join. Sul fronte dei sistemi multimediali, risulta necessario disporre di strutture di memorizzazione e accesso specifiche.

Esistono due approcci nella introduzione degli oggetti alle basi di dati. Le "basi di dati orientate a oggetti" (*Object-Oriented Database Management Systems*, OODBMS) hanno seguito un approccio più rivoluzionario, estendendo i DBMS a partire dalle caratteristiche dei linguaggi di programmazione a oggetti. I sistemi "relazionali a oggetti" (*Object-Relational Database Management Systems*, ORDBMS) hanno viceversa assunto un approccio più evolutivo, integrando il concetto di oggetto all'interno del modello relazionale. Va osservato che i due approcci, apparentemente conflittuali all'inizio degli anni Novanta, si sono poi dimostrati nei fatti abbastanza convergenti.

In questo capitolo focalizzeremo l'attenzione sui sistemi a oggetti, presentando i tipici componenti di un modello a oggetti e mostrando poi le caratteristiche degli OODBMS e degli ORDBMS. Discuteremo poi alcuni aspetti tecnologici relativi all'implementazione di questi sistemi. Descriveremo quindi i sistemi multimediali, che possono anche essere considerati un caso particolare di sistema a oggetti. È bene osservare che tra le soluzioni di gestione dei dati che vanno oltre il modello relazionale rientrano anche i sistemi per la gestione di XML descritti nel Capitolo 13, le basi di dati semantiche trattate nel Capitolo 14 e i sistemi NoSQL descritti nel Capitolo 19.

20.1 Basi di dati a oggetti

In questo paragrafo introduciamo progressivamente i concetti necessari per descrivere una base di dati a oggetti; essi estendono notevolmente il potere espressivo del modello dei dati rispetto alla spartana semplicità del modello relazionale, e ritroviamo nel modello dei dati molte delle caratteristiche del modello Entità-Relazione, visto nel Capitolo 6.

20.1.1 Tipi

I tipi, in un linguaggio di programmazione a oggetti, consentono di definire le proprietà degli oggetti; in particolare, distinguiamo proprietà *statiche* (che descrivono la struttura

degli oggetti) e *dinamiche* (che descrivono le operazioni, o "metodi", applicabili agli oggetti).

Partiamo, nella descrizione dei tipi, dalla parte statica; la natura dinamica dei tipi verrà illustrata nel Paragrafo 20.1.3. La parte statica di un tipo è costruita usando i cosiddetti *costruttori di tipo* e un insieme abbastanza esteso di *tipi di dati atomici*, che comprendono i classici tipi di dati presenti nei linguaggi di programmazione: per esempio, interi, reali, booleani, stringhe di caratteri. I tipi atomici includono gli identificatori di oggetto (*Object Identifier*, OID), che verranno introdotti più avanti in questo capitolo. La maggior parte dei sistemi include il valore nullo (`nil`) in tutti i tipi atomici; si dice che `nil` è un valore *polimorfo*, che appartiene cioè a molti tipi.

Ogni definizione di tipo associa un nome (etichetta) a un tipo; per esempio, `Indirizzo: string` è una definizione di tipo che associa il nome "Indirizzo" al tipo atomico `string`.

Tipi di dati complessi I costruttori di tipo consentono di costruire tipi, detti *tipi di dati complessi*, che descrivono la struttura delle istanze (anche dette *oggetti complessi* di una base di dati a oggetti). Una definizione ricorsiva dei tipi di dati complessi, basata sui costruttori di tipo, è la seguente. Assumiamo come dato un insieme di tipi di dati atomici.

- Il costruttore *record* consente di costruire tipi le cui istanze sono tuple di valori (complessi) di tipi differenti. Se $T_1, \ldots, T_n$ sono nomi di tipo e $A_1, \ldots, A_n$ sono etichette distinte, che chiameremo *attributi* del record, $T = \textit{record-of}(A_1 : T_1, \ldots, A_n : T_n)$ è un *tipo record*.
- I costruttori di *insieme*, *multi-insieme* e *lista* consentono di costruire collezioni di valori (eventualmente complessi) dello stesso tipo. Gli insiemi (*set*) sono collezioni non ordinate e prive di duplicati, i multi-insiemi (*bag*) sono collezioni non ordinate che possono presentare elementi duplicati, le liste (*list*) sono collezioni ordinate che possono presentare elementi duplicati. Se T_1 è un tipo, $T = \textit{set-of}(T_1)$ è un *tipo insieme*, $T = \textit{bag-of}(T_1)$ è un *tipo multi-insieme* e $T = \textit{list-of}(T_1)$ è un *tipo lista*.

Dato un tipo complesso T, un oggetto che ha per tipo T si dice *istanza* di T. Come è consuetudine in molte basi di dati a oggetti, assumiamo che una definizione di tipo di dato inizi sempre con un costruttore di tipo record. Dato un oggetto x di tipo $T = \textit{record-of}(A_1 : T_1, \ldots, A_n : T_n)$, possiamo in tal modo parlare degli attributi $A_1, \ldots, A_n$ come *proprietà* di x.

L'uso dei costruttori di tipo garantisce la cosiddetta *complessità strutturale* degli oggetti; in particolare, se un oggetto del mondo reale è complesso, i costruttori di tipo ci consentono di modellarlo in modo accurato. Alcuni sistemi non supportano però tutti i costruttori, e d'altra parte non è in genere conveniente costruire tipi eccessivamente complessi, perché poi diventa difficile accedere, tramite linguaggi di programmazione e interrogazione, ai componenti del tipo.

Vediamo un esempio di definizione di tipo per l'oggetto complesso Automobile, caratterizzato da varie proprietà: Targa, Modello, Produttore, Colore, Prezzo, PartiMeccaniche. Alcune di queste proprietà hanno a loro volta una struttura complessa:

```
Automobile: record-of(
              Targa: string,
              Modello: string,
              Produttore: record-of(
                          Nome: string,
                          Presidente: string,
                          Stabilimenti: set-of(
                            record-of(
                               Nome: string,
```

```
                                Città: string,
                                Addetti: integer))),
                Colore: string,
                Prezzo: integer,
                PartiMeccaniche: record-of(
                            Motore: string,
                            Ammortizzatore: string))
```

Data questa definizione di tipo, è possibile esemplificare valori che sono compatibili con essa. Per esempio, assumendo una rappresentazione dei valori in cui i record sono contenuti all'interno di parentesi quadre e gli insiemi all'interno di parentesi graffe, un valore compatibile con la definizione precedente è:

```
V1: ["DB123MS", "Punto",
       ["Fiat", "Agnelli",
          {["Mirafiori","Torino", 10000],
           ["trattori", "Modena", 1000]}],
     "blu", 9500, ["1100cc", "A olio"]]
```

Dato un valore di tipo record, è possibile accedere ai suoi componenti tramite la classica *notazione punto* (*dot notation*), che può essere applicata in modo ricorsivo. Per esempio:

```
V1.Colore="blu"
V1.Produttore.Presidente="Agnelli"
V1.PartiMeccaniche.Ammortizzatore="A olio"
```

Oggetti e valori La possibilità di introdurre un'arbitraria complessità strutturale, illustrata da questo esempio, soddisfa l'esigenza di associare a un unico oggetto una struttura arbitrariamente complessa; quindi, un automezzo (oppure un circuito integrato) viene descritto in modo più articolato e unitario che non, per esempio, usando il modello relazionale e separandone la descrizione in tante tabelle. Questo esempio illustra, però, anche il principale limite di una descrizione strutturale basata su "valori": per ogni automobile, per esempio della Fiat, viene ripetuta la descrizione del costruttore, che a sua volta si compone di vari dati, tra cui il nome del presidente e l'insieme degli stabilimenti. Tale descrizione introduce ovviamente della ridondanza, e contraddice i principi di normalizzazione dei dati, presentati nel Capitolo 9.

Per ovviare a questo problema è necessario decomporre la struttura precedente; per rappresentare i legami tra i diversi frammenti si usano i riferimenti fra oggetti (OID). La parte strutturale di un oggetto è in realtà costituita da una coppia (OID, *Valore*); il valore è un'istanza del tipo dell'oggetto; lo chiamiamo lo "stato" dell'oggetto. L'OID garantisce l'individuazione in modo univoco dell'oggetto nella base di dati e consente di costruire riferimenti fra oggetti; è assegnato dal DBMS e in molti sistemi non è visibile all'utente. Un oggetto può includere riferimenti espliciti ad altri oggetti; denotiamo ciò, a livello di schema, usando la notazione $*T$, che denota gli OID di oggetti di tipo T. Se una proprietà di un oggetto ha tipo $*T$, si dice che essa ha *per valore un oggetto* (è *object-valued*). La seguente definizione introduce riferimenti a oggetti:

```
Automobile: record-of(Targa: string,
                      Modello: string,
                      Produttore: *Costruttore,
                      Colore: string,
                      Prezzo: integer,
                      PartiMeccaniche: record-of
                          (Motore: string,
                           Ammortizzatore: string))
```

```
Costruttore: record-of(Nome: string,
                       Presidente: string,
                       Stabilimenti:
                            set-of(*Stabilimento))
Stabilimento: record-of(Nome: string,
                        Città: string,
                        Addetti: integer)
```

Un insieme di oggetti compatibili con le definizioni di tipo appena introdotte è il seguente:

```
01:  <OID1, ["DB123MS", "Punto", OID2, "blu", 9500,
           ["1100cc", "A olio"]]>
02:  <OID2, ["Fiat","Agnelli", {OID3,OID4}]>
03:  <OID3, ["Mirafiori", "Torino", 10000]>
04:  <OID4, ["trattori", "Modena", 1000]>
```

L'esempio mostra che le proprietà object-valued consentono riferimenti fra oggetti (da una automobile al suo costruttore, dal costruttore ai suoi stabilimenti) e la condivisione di oggetti da parte di altri oggetti (lo stesso costruttore viene referenziato da varie automobili). Per esempio:

- il valore `01.Produttore` è l'OID dell'oggetto `02`;
- il valore `01.Produttore.Presidente` è la stringa `"Agnelli"`.

Identità e uguaglianza L'uso di OID consente anche di garantire la possibilità che due oggetti distinti abbiano lo stesso stato e differiscano solo per l'OID (per esempio, due automobili con le stesse proprietà); questa possibilità non è consentita dal modello relazionale. Diremo quindi che due oggetti `01` e `02` sono identici (`01=02`) quando hanno lo stesso identificatore (e ovviamente anche lo stesso stato), e utilizzeremo la nozione di uguaglianza per confrontare lo stato di due oggetti.

Nel modello a oggetti, esistono due nozioni di uguaglianza:

- l'*uguaglianza superficiale* (==) richiede che due oggetti abbiano lo stesso stato.
- l'*uguaglianza profonda* (===) richiede che due oggetti abbiano identici valori ottenuti sostituendo ricorsivamente gli oggetti raggiungibili tramite OID agli OID stessi, ma non necessariamente lo stesso stato.

Si noti che lo stato include le proprietà object-valued (OID), perciò l'uguaglianza superficiale implica l'uguaglianza profonda; si noti inoltre che la procedura indicata per costruire i valori raggiungibili nella definizione di uguaglianza profonda deve essere costruita con attenzione, altrimenti potrebbe non terminare in presenza di riferimenti ciclici. In genere, i sistemi offrono solo un operatore per verificare l'uguaglianza superficiale di due oggetti, mentre l'uguaglianza profonda deve essere programmata tramite opportuni predicati sui valori. Per esempio, si considerino le seguenti definizioni di tipo e i seguenti oggetti:

```
T1: record-of(A: integer, B: *T2)
T2: record-of(C: character, D: *T3)
T3: record-of(E: integer)

01: <OID1, [120, OID4]> di tipo T1
02: <OID2, [120, OID4]> di tipo T1
03: <OID3, [120, OID5]> di tipo T1
04: <OID4, ["a", OID6]> di tipo T2
05: <OID5, ["a", OID7]> di tipo T2
06: <OID6, [15]> di tipo T3
07: <OID7, [15]> di tipo T3
```

In questo caso:

- le uguaglianze superficiali sono: `01==02`, `06==07`;
- le uguaglianze profonde sono: `01===02`, `01===03`, `02===03`, `04===05`, `06===07`;
- la condizione per definire l'uguaglianza profonda di oggetti `X` e `Y` di tipo `T1`, che può essere programmata, è:

```
X.A=Y.A and X.B.C=Y.B.C and X.B.D.E==Y.B.D.E.
```

20.1.2 Classi

Una classe raccoglie oggetti dello stesso tipo; essa funge cioè da contenitore di oggetti, che possono essere dinamicamente aggiunti e tolti alla classe. Gli oggetti che appartengono alla stessa classe sono omogenei, sono cioè dotati dello stesso tipo. Nel DDL, le definizioni di tipo possono essere talvolta date nell'ambito della definizione della classe. In genere, la definizione di una classe è separata in due parti.

- L'*interfaccia* descrive il tipo statico e dinamico degli oggetti appartenenti alla classe; il tipo dinamico include la segnatura di tutti i metodi della classe. Ogni segnatura contiene l'elenco del nome e del tipo dei parametri del metodo, che possono essere di input e/o di output; la conoscenza della segnatura consente di invocare il metodo da un programma.
- L'*implementazione* descrive il codice dei metodi e, talvolta, le strutture dati per memorizzare gli oggetti.

Affronteremo la descrizione del tipo dinamico e dell'implementazione nel Paragrafo 20.1.3, dedicato ai metodi.

Il principio di *incapsulamento*, un'importante astrazione dei linguaggi a oggetti, deriva dal più generale concetto di *tipo di dato astratto*, che attribuisce a ciascun oggetto un'interfaccia e un'implementazione. L'interfaccia descrive solo le operazioni applicabili sull'oggetto, mentre l'implementazione nasconde la struttura dati e l'effettiva costruzione delle operazioni. Nelle basi di dati a oggetti, tuttavia, lo stato è spesso visibile tramite alcune interfacce utente (per esempio, dal linguaggio di interrogazione); pertanto, rispetto a un'interpretazione rigorosa di tipo di dato astratto, le basi di dati a oggetti "espongono" ai loro utenti la struttura dello stato degli oggetti.

La distinzione fra tipi e classi è uno degli argomenti più controversi nell'ambito dei linguaggi di programmazione e delle basi di dati a oggetti; nel nostro modello dei dati, i tipi sono astrazioni che consentono di descrivere sia lo stato sia il comportamento, mentre le classi descrivono sia la rappresentazione estensionale degli oggetti, sia l'implementazione dei metodi relativi a un tipo; il tipo, cioè, descrive proprietà astratte, mentre la classe descrive la realizzazione di tali proprietà astratte tramite strutture dati e programmi. Abbiamo quindi presentato un modello dei dati in cui:

- tipi e classi sono concetti distinti;
- ogni classe è associata a un solo tipo;
- il concetto di classe descrive sia l'implementazione sia l'estensione di un tipo.

Il rapporto che intercorre tra valori, oggetti, tipi e classi è sintetizzato in Figura 20.1. Ogni oggetto ha un valore, che appartiene a un tipo; ogni oggetto appartiene a una classe, che è associata a un tipo; il tipo della classe è il tipo cui appartiene il valore degli oggetti contenuti nella classe.

Modelli dei dati più complessi riservano al concetto di classe il solo ruolo di definire l'implementazione dei metodi, introducendo poi un terzo concetto, quello di *estensione*, che consente di inserire oggetti dello stesso tipo in collezioni differenti e dare nomi distinti a queste collezioni (per esempio, il tipo Cittadino potrebbe

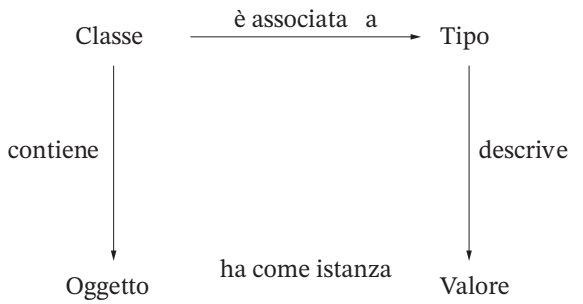

Figura 20.1
Relazione fra valori, oggetti, tipi e classi.

corrispondere a una medesima classe ma a collezioni differenti, denominate Milanese e Fiorentino). In tal caso sarebbero presenti, ben distinti, i tre concetti di tipo, classe ed estensione. D'altra parte, alcuni modelli dei dati più semplici non distinguono neppure tra tipi e classi, in quanto fanno coincidere i due concetti e attribuiscono al tipo anche il ruolo di definire estensioni e implementazioni dei metodi. Per esempio, vediamo come la definizione di classe includa sintatticamente anche la definizione di tipo; si noti l'uso di *nomi di classi* nella definizione di tipo, che vengono implicitamente interpretati come riferimenti:[1]

```
add class Automobile
  type tuple(Targa: string,
             Modello: string,
             Produttore: Costruttore,
             Colore: string,
             Prezzo: integer,
             PartiMeccaniche:
                  tuple(Motore: string,
                        Ammortizzatore: string))
add class Costruttore
  type tuple(Nome: string,
             Presidente: Persona,
             Stabilimenti:
                  set(Stabilimento))

                    add class Stabilimento
                      type tuple(Nome: string,
                                 Città: string,
                                 Addetti: integer)

                    add class Persona
                      type tuple(Nome: string,
                                 Residenza: string,
                                 CodFisc: string)
```

La struttura delle classi può essere rappresentata in modo grafico, mettendo in evidenza i legami fra le classi dovuti a proprietà object-valued. La Figura 20.2 mostra le quattro classi fin qui introdotte, inserite in uno schema che comprende anche altre classi e una gerarchia di generalizzazione, illustrata nel seguito. In aggiunta, la figura illustra una classe Persona, cui fanno riferimento sia l'attributo **Presidente** nella classe Costruttore, sia l'attributo **Pilota** introdotto nella classe AutoSportiva.

[1] La sintassi che verrà utilizzata per presentare le caratteristiche strutturali delle classi deriva dal sistema O2, una base di dati a oggetti che ha avuto una particolare influenza nello sviluppo di questa tecnologia, ma che non è più disponibile in commercio.

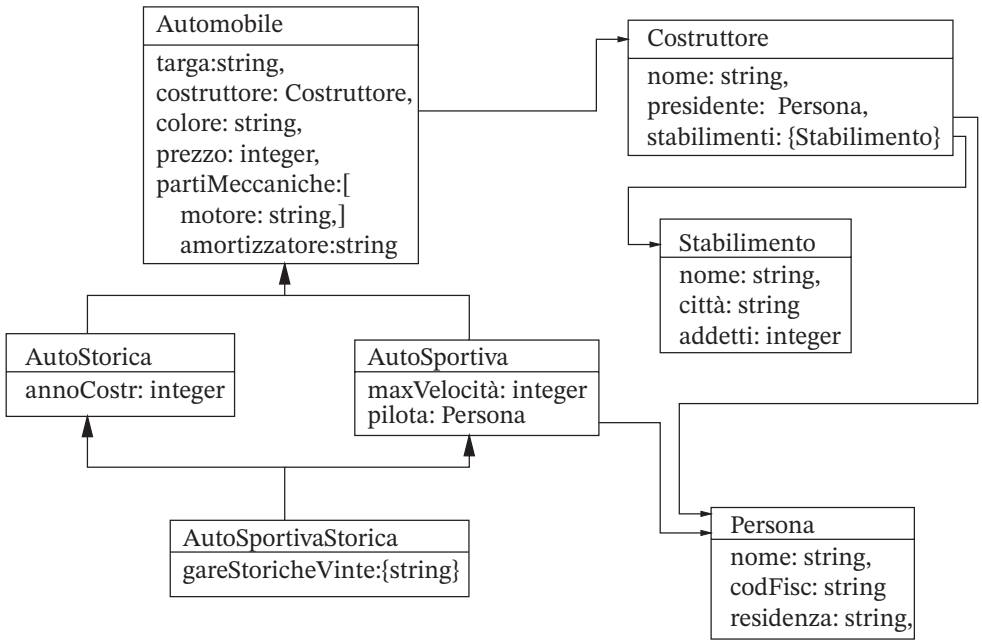

Figura 20.2 Schema di una base di dati a oggetti per la descrizione di automobili.

20.1.3 Metodi

I metodi vengono usati per manipolare gli oggetti di un OODBMS; la loro presenza è il principale elemento di innovazione di un OODBMS rispetto a una base di dati relazionale. Un metodo ha una *segnatura*, che ne descrive l'interfaccia e comprende tutte le informazioni che consentono di invocarlo, e un'*implementazione*, che contiene il codice del metodo; spesso le implementazioni dei metodi sono scritte in linguaggi di programmazione a oggetti, quali C++, Java e C#. Più propriamente, come già discusso, la segnatura del metodo è una delle componenti della definizione di tipo, mentre l'implementazione del metodo è una delle componenti della definizione di classe.

Ciascun metodo è associato con una specifica classe di oggetti. Assumeremo che ogni metodo abbia un numero arbitrario di *parametri di ingresso* e un unico *parametro di uscita*.

I metodi presenti in un OODBMS possono essere classificati in quattro categorie:

- i *costruttori* sono utilizzati per costruire gli oggetti a partire da parametri di ingresso;
- i *distruttori* servono per cancellare gli oggetti ed eventuali altri oggetti a essi collegati;
- gli *accessori* restituiscono informazioni sul contenuto degli oggetti;
- i *trasformatori* cambiano il contenuto dello stato degli oggetti.

Altri metodi non possono essere classificati in base a questo schema, e rispondono a specifiche esigenze applicative.

In molti sistemi si distingue tra metodi *pubblici* e *privati*; i metodi pubblici sono richiamabili in qualunque punto del programma applicativo, mentre i metodi privati sono richiamabili solo all'interno degli altri metodi della stessa classe. In base a questa distinzione, gli oggetti risultano massimamente incapsulati quando è possibile accedere loro soltanto tramite i metodi pubblici.

La segnatura può far parte sintatticamente della definizione della classe, oppure ciascun metodo può essere introdotto da una definizione specifica, in cui si nomina la classe cui appartiene. Questa seconda opzione consente una definizione incrementale dello schema, e verrà utilizzata nel seguito. Il metodo init è un trasformatore e inizializza alcune delle proprietà delle Automobili; esso riceve come parametri di

ingresso alcuni valori con cui inizializzare lo stato di un oggetto, e restituisce come parametro di uscita l'oggetto stesso. Ogni metodo si applica a un oggetto della classe Automobile, che si può perciò considerare un parametro di ingresso implicitamente definito. Vediamo la definizione della segnatura (istruzione `add method`):

```
add method init(Targa_par: string,
                Modello_par: string,
                Colore_par: string,
                Prezzo_par: integer): Automobile
           in class Automobile is public
```

Vediamo ora l'implementazione nella sintassi del linguaggio C++:

```
Ref<Automobile> Automobile::init
         (string Targa_par, string Modello_par,
          string Colore_par,  integer Prezzo_par)
{   self -> Targa = Targa_par;
    self -> Modello = Modello_par;
    self -> Colore = Colore_par;
    self -> Prezzo = Prezzo_par;
    return(self); }
```

Le implementazioni dei metodi sono scritte in C++; per l'integrazione tra le basi di dati a oggetti e il linguaggio C++, facciamo riferimento alla proposta del consorzio ODMG. L'espressione `Ref<Automobile>` dichiara che il tipo del risultato dell'invocazione del metodo è un riferimento a un oggetto della classe Automobile; il costrutto `Ref`, che sfrutta un apposito meccanismo del linguaggio C++ (i *template*), consente di definire all'interno del programma C++ il tipo degli OID degli oggetti conservati nella base di dati. I riferimenti possono essere utilizzati come i puntatori nell'ambito del programma. L'espressione `Automobile::init` specifica invece che si sta definendo l'implementazione del metodo **init** della classe Automobile. La variabile `self`, introdotta implicitamente nell' implementazione, denota l'oggetto cui viene applicato il metodo; con altra terminologia, `self` denota l'oggetto *ricevente*, cioè quello al quale viene inviato un *messaggio*. La notazione punto, introdotta nel Paragrafo 20.1.1, è resa con i meccanismi di accesso propri del C++, per cui la composizione dell'operatore che segue un riferimento (`*`) e dell'operatore che accede a un attributo (`.`) viene rappresentata con una freccia (`->`). L'invocazione del metodo **init** in un programma avviene nel modo seguente:

```
Ref<Automobile> X;
X = new(Automobile);
X -> init("DB313MS", "Panda", "blu", 7000);
```

La prima istruzione del programma definisce una variabile di nome `X` e di tipo `Ref<Automobile>`. La seconda istruzione crea un oggetto della classe Automobile, tramite l'invocazione del metodo **new**, e lo rende accessibile tramite la variabile `X`; il metodo polimorfo **new** è offerto da tutte le classi per creare nuovi oggetti e inserirli nella classe. Infine, la terza istruzione applica il metodo **init** all'oggetto, inizializzandone alcune proprietà. Si noti che nella chiamata di un metodo si deve indicare l'oggetto target (o ricevente) e il nome del metodo, seguito dall'elenco dei valori attuali dei parametri di ingresso. Al termine dell'esecuzione del metodo, viene restituito come parametro di uscita l'identificatore dell'oggetto su cui il metodo stesso è invocato.

Il metodo **Aumento** nella classe Automobile è un *trasformatore*; esso incrementa il **Prezzo** di un certo `Ammontare`; `Ammontare` è l'unico parametro del metodo.

```
add method Aumento(Ammontare:integer)
    in class Automobile is public
...
void Automobile::Aumento(Ammontare:integer)
{ self -> Prezzo += Ammontare;}
```

Il prossimo esempio mostra l'invocazione innestata dei metodi **init** e **Aumento**, possibile in quanto il metodo **init** restituisce come parametro di uscita l'oggetto target:[2]

```
Ref<Automobile> X;
X = new(Automobile);
(X -> init("DB313MS", "Panda", "blu", 7000))
   -> Aumento(1200);
```

Possiamo infine riassumere le proprietà degli oggetti. Ogni oggetto ha un identificatore, uno stato e un comportamento. L'*identificatore* garantisce l'individuazione in modo univoco dell'oggetto nella base di dati, e consente di costruire riferimenti fra oggetti. Lo *stato* di un oggetto è l'insieme dei valori assunti dalle sue proprietà in un determinato istante. Infine, il *comportamento* di un oggetto è definito dai metodi che possono essere applicati all'oggetto stesso, e ne predefiniscono l'evoluzione nel tempo.

Conflitto di impedenza Gli esempi di metodi appena visti consentono di introdurre una discussione relativa a una caratteristica importante delle basi di dati a oggetti, nelle quali il programmatore può manipolare oggetti persistenti tramite le istruzioni del linguaggio di programmazione. Si dice che i DBMS a oggetti risolvono il cosiddetto *conflitto d'impedenza* (*impedance mismatch*) che caratterizza invece i sistemi relazionali; esso consiste nella necessità di far dialogare un linguaggio di programmazione, contenente variabili scalari, con il linguaggio SQL, che estrae insiemi di tuple. In effetti, i meccanismi di interazione fra l'ambiente di programmazione e i server SQL, basati sull'uso di cursori per scandire il risultato delle interrogazioni (descritti nel Capitolo 10), sono assai rigidi e poco "amichevoli"; viceversa, nella programmazione degli OODBMS gli oggetti persistenti e temporanei possono essere gestiti con semplicità (per esempio, tramite meccanismi di assegnamento). Si dice in tal caso che la persistenza è una caratteristica *ortogonale*, di cui il programmatore può essere inconsapevole.

20.1.4 Gerarchie di generalizzazione

La possibilità di stabilire gerarchie di generalizzazione fra classi è probabilmente l'astrazione più importante nei linguaggi e nelle basi di dati a oggetti. Le generalizzazioni consentono la definizione di una sotto-classe a partire da una super-classe; la sotto-classe *eredita* lo stato e il comportamento della super-classe, e può in aggiunta rendere il proprio stato e comportamento più specifici tramite l'aggiunta di attributi e metodi. Le gerarchie di generalizzazione fra classi sono del tutto analoghe alle gerarchie di generalizzazione fra entità offerte nel modello Entità-Relazione; esse garantiscono la cosiddetta *complessità semantica* degli oggetti. In una gerarchia di generalizzazione:

- tutti gli oggetti delle sotto-classi appartengono automaticamente alle super-classi;
- tutti gli attributi e i metodi delle super-classi vengono *ereditati* dalle sotto-classi;
- è possibile introdurre nella descrizione delle sotto-classi dei nuovi attributi e dei nuovi metodi.

[2] Si noti che l'esempio non è molto sensato sul piano applicativo, perché prima inizializza e poi modifica la proprietà **Prezzo**, dandone un immediato aumento.

È possibile ridefinire l'implementazione di un metodo senza modificare la sua interfaccia, in modo da ottenere varie implementazioni dello stesso metodo, richiamabili in modo uniforme su oggetti di tipo diverso appartenenti alla gerarchia: il sistema utilizza l'implementazione più specifica in base al tipo dell'oggetto; affronteremo questo aspetto nel Paragrafo 20.1.6. È poi possibile, anche se ciò pone dei problemi, *raffinare* (cioè rendere più specifici) gli attributi e i metodi presenti nelle super-classi; affronteremo questo aspetto nel Paragrafo 20.1.7.

Le generalizzazioni godono della proprietà transitiva; perciò, se C_1 è una sotto-classe di C_2 e C_2 è una sotto-classe di C_3, anche C_1 è una sotto-classe di C_3. La relazione di sotto-classe deve essere aciclica.

In virtù della proprietà di ereditarietà, gli attributi e i metodi definiti nell'ambito di tutte le super-classi sono automaticamente ereditati dalle sotto-classi, le cui definizioni possono pertanto limitarsi a introdurre nuovi attributi e metodi. Per esempio, definiamo le sotto-classi AutoSportiva e AutoStorica della classe Automobile:

```
add class AutoSportiva
  inherits Automobile
  type tuple(MaxVelocita:integer,
             Pilota:Persona)

add class AutoStorica
  inherits Automobile
  type tuple(AnnoCostr:integer)
```

In virtù della ereditarietà, la classe AutoStorica eredita gli attributi e i metodi definiti per Automobile (per esempio, gli attributi Modello e Colore e i metodi init e Aumento). È possibile quindi invocare il metodo init su un oggetto della classe AutoStorica:

```
Ref<AutoStorica> X;
X = new(AutoStorica);
X -> init("Mi56543", "Ferrari", "rossa", 170000);
X -> AnnoCostr = 1957;
```

L'implementazione di un metodo *m* richiamabile in una classe C_1 può non essere definita nella classe C_1; in tal caso, è necessario che un'implementazione di *m* sia definita per una super-classe C_2 di C_1; nel caso vi siano due super-classi C_2 e C_3 della classe C_1 che possiedono un'implementazione di *m*, si sceglie l'implementazione nella classe inferiore in gerarchia, cioè l'implementazione più specifica rispetto alla classe C_1.

Migrazioni fra classi In presenza di gerarchie di generalizzazione, in alcuni OODBMS gli oggetti possono migrare da un livello della gerarchia a un altro; in altri OODBMS, invece, gli oggetti rimangono nella classe in cui sono creati durante tutta la loro esistenza. Si chiama *specializzazione* l'operazione tramite la quale un oggetto migra da una super-classe a una sotto-classe; in questo "cammino", lo stato dell'oggetto in genere si modifica, aggiungendo nuove proprietà. La migrazione inversa è detta *generalizzazione* e consente a un oggetto di migrare da una sotto-classe a una super-classe; lo stato, in genere, perde alcune delle sue proprietà.

Per esempio, un oggetto della classe Automobile può essere specializzato, entrando a far parte della classe AutoStorica, a un certo punto della sua esistenza; viceversa, un oggetto della classe AutoSportiva cui è applicata l'operazione di generalizzazione cessa di appartenere a tale classe, rimanendo però nella classe Automobile.

Distinguiamo tra *istanze* e *membri* di una classe; un oggetto è istanza di una classe solo se essa è la *classe più specializzata* per l'oggetto nell'ambito di una gerarchia di

generalizzazione; le istanze di una classe sono automaticamente membri delle sue super-classi. In alcuni sistemi, ogni oggetto può essere un'istanza di una sola classe; in altri, invece, alcuni oggetti possono essere istanze di più classi, cioè appartenere a due o più distinte classi più specializzate, tra loro incomparabili dal punto di vista della gerarchia. Nel nostro esempio, un oggetto della classe Automobile può essere istanza di due classi se esso viene specializzato sia nella classe AutoSportiva sia nella classe AutoStorica.

Ereditarietà multipla In alcuni sistemi è consentito a una classe di ereditare da più super-classi; questa situazione si dice *ereditarietà multipla*. Per esempio, è possibile definire la classe AutoSportivaStorica, nel modo seguente:

```
add class AutoSportivaStorica
  inherits AutoSportiva, AutoStorica
  type tuple(GareStoricheVinte: set(string))
```

Si noti che questa gerarchia di classi definisce una situazione di appartenenza alle classi, illustrata in Figura 20.3, in cui:

- le istanze della classe AutoSportivaStorica sono automaticamente membri delle classi Automobile, AutoSportiva e AutoStorica;
- le istanze della classe AutoSportiva o AutoStorica che non fanno parte della classe AutoSportivaStorica sono automaticamente membri della classe Automobile;
- infine, esistono istanze della classe Automobile che non fanno parte delle classi AutoSportiva o AutoStorica.

Si noti che, qualora il sistema ammetta oggetti che sono contemporaneamente istanze delle due classi AutoSportiva e AutoStorica (come classi più specializzate), essi possono esistere nella base di dati *senza* far parte della classe AutoSportivaStorica; l'appartenenza di un oggetto a una classe richiede infatti un'esplicita operazione di inserimento di quell'oggetto nella classe. Questo fatto è illustrato in Figura 20.3 tramite una rappresentazione della classe AutoSportivaStorica come sotto-insieme dell'intersezione delle classi AutoSportiva e AutoStorica.

Conflitti L'ereditarietà multipla oppure la presenza di oggetti che sono istanze di molteplici classi possono essere la fonte di *conflitti di nome*, qualora due o più super-classi abbiano attributi o metodi con lo stesso nome. In tal caso, occorre definire delle politiche per la risoluzione del conflitto, in modo da rendere non ambiguo il meccanismo di ereditarietà. Alcune delle soluzioni possibili sono elencate di seguito.

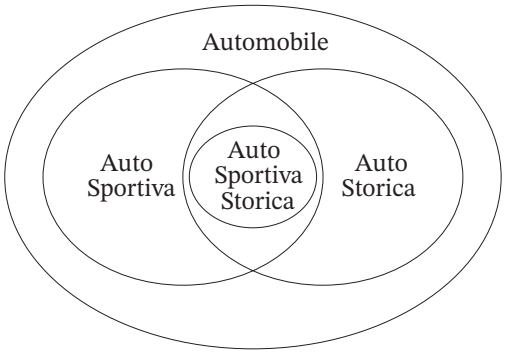

Figura 20.3 Rappresentazione degli oggetti appartenenti alle classi Automobile, AutoSportiva, AutoStorica e AutoSportivaStorica.

- Rilevare il conflitto all'atto della definizione delle classi e non accettare come corrette le definizioni. Se si esclude la possibilità di molteplici istanze per un oggetto, è sufficiente fare questa verifica all'atto della definizione di classi con ereditarietà multipla. Questa soluzione ha lo svantaggio di imporre un ripensamento dello schema relativamente a parti già consolidate, in quanto il problema si elimina solo cambiando la struttura delle classi che sono causa di conflitto o rinunciando alla definizione della sotto-classe.
- Definire dei meccanismi per togliere ogni ambiguità dalla scelta; per esempio, utilizzando un ordinamento fra le classi definito a priori, oppure applicando il metodo nel contesto di una variabile target che sia stata esplicitamente definita membro di una sola delle classi sulle quali sono definiti gli attributi e i metodi che danno luogo a problemi di ambiguità.
- Ridefinire gli attributi e i metodi localmente, come descritto nel Paragrafo 20.1.6; la ridefinizione locale, infatti, elimina la visibilità del conflitto.

20.1.5 Persistenza

Gli oggetti definiti in un programma possono essere *persistenti* oppure *temporanei*; gli oggetti temporanei cessano di esistere al termine dell'esecuzione del programma, mentre gli oggetti persistenti vengono inseriti nell'OODB. In genere, un oggetto diviene persistente tramite i seguenti meccanismi:

- tramite l'inserimento in una classe che è definita come persistente. Si adopera in tal caso la primitiva new, come descritto in alcuni esempi di questo paragrafo;
- tramite la *raggiungibilità* a partire da un altro oggetto persistente; per esempio, se vengono creati oggetti di tipo Automobile e di tipo Costruttore in cui il primo fa riferimento al secondo e poi il primo viene inserito in una classe persistente, anche il secondo diviene persistente. In tal modo, lo stato di un oggetto ricostruibile ricorsivamente tramite riferimenti è persistente;
- tramite la *denominazione*, cioè dando a un oggetto un nome (detto *handle*, cioè letteralmente "maniglia" di un oggetto) che possa essere utilizzato per ritrovarlo a una successiva invocazione del programma. Nell'esempio, usando l'interfaccia di integrazione per C++ proposta dall'ODMG, è possibile dichiarare una variabile e poi darle un nome, rendendo il corrispondente oggetto persistente:

```
Ref<AutoSportiva> X;
MioDB->set_object_name(X,"Ferrari_360_Modena");
```

Non tutti questi meccanismi per dare la persistenza agli oggetti sono supportati in ogni OODB. In alcuni sistemi, si definisce la persistenza a livello di classe, introducendo cioè classi persistenti e classi temporanee; la natura persistente o temporanea degli oggetti viene definita quando l'oggetto è inserito nella classe.

La persistenza tramite raggiungibilità garantisce la cosiddetta "integrità referenziale" dell'OODB, cioè il mantenimento automatico di vincoli di integrità referenziale fra le classi che sono simili ai vincoli di riferimento fra tabelle, discussi nel Capitolo 2. Tuttavia questo tipo di persistenza comporta una qualche difficoltà nella cancellazione degli oggetti dall'OODB. In pratica, un oggetto può essere cancellato dal sistema solo quando esso non è più raggiungibile tramite denominazione o tramite riferimenti da altri oggetti persistenti; questo processo si chiama *garbage collection*.

20.1.6 Ridefinizioni dei metodi

Una volta introdotta una gerarchia, è possibile ridefinire i metodi delle sotto-classi; questa tecnica, detta *overriding* dei metodi, è estremamente utile per garantire una

interfaccia uniforme dei metodi, applicabile indipendentemente dall'appartenenza di un oggetto a uno specifico elemento della gerarchia delle classi, e un'implementazione che tiene viceversa conto della classe di appartenenza degli oggetti.

L'esempio classico che viene utilizzato per introdurre questo approccio consiste nel metodo display, che fornisce una rappresentazione dell'oggetto; supponiamo l'esistenza di una generica classe Oggetto, da cui tutte le altre classi ereditano, in cui è definita l'interfaccia del metodo display, che non abbia parametri espliciti di input (l'oggetto target è un parametro implicito) e sia associata a un'implementazione fittizia. Il metodo viene poi ridefinito nell'ambito di tutte le altri classi, che ereditano da Oggetto; avremo per esempio le classi Proprietario, Abitazione, Mappa, CondizioniVendita relative alle attività di un'agenzia immobiliare. Nell'ambito di ciascuna classe, il metodo display viene implementato in modo differente; per esempio, display applicato all'abitazione mostra una sua fotografia, alla mappa mostra una planimetria dell'abitazione, al proprietario e alle condizioni di vendita mostra una schermata con i dati relativi. È possibile cioè scrivere del codice che, in modo compatto, dato un insieme anche eterogeneo di oggetti S, richiama la funzione display su di essi; per esempio:

```
for X in S do X.display()
```

Ovviamente, la scelta di quale metodo invocare dipende dal tipo dell'oggetto cui il metodo si applica; in particolare, se il tipo dell'oggetto non può essere noto a tempo di compilazione (per esempio, perché gli oggetti possono migrare fra le classi), tale scelta deve avvenire a tempo di esecuzione. Questa caratteristica è detta *late binding* (la traduzione letterale è: "collegamento tardivo") e comporta, da parte del sistema, la scelta del codice specifico per l'esecuzione del metodo e il suo collegamento con il resto del codice applicativo a tempo di esecuzione.

Per effetto della ridefinizione, è possibile avere quindi varie versioni dello stesso metodo con identica interfaccia (e, in particolare, identico nome); questo fenomeno si chiama *overloading* dei nomi di metodi.

Vediamo un esempio di uso di metodo con overloading e overriding. Si consideri la gestione di dati per assistere il progetto di sistemi software. In tal caso, introduciamo una gerarchia di generalizzazione con la classe generica File e le sotto-classi Sorgente e Documentazione. Queste classi sono caratterizzate da attributi introdotti localmente; il metodo init è in grado di inizializzare gli oggetti in base al loro tipo.

```
add class File
type tuple(Nome: string,
           Creatore: Utente,
           Data: date)
method init(Nome_par: string) is public

add class Sorgente
    inherits File
type tuple(Responsabile: Utente)

add class Documentazione
    inherits File
type tuple(Validità: date)
...

Ref<File> File::init(string Nome_par)
{     self -> Nome = Nome_par;
      self -> Creatore = DatabaseUser();
      self -> Data = Today();
      return(self);}
```

```
Ref<Sorgente> Sorgente::init(string Nome_par)
{     (Ref<File>) self -> init(Nome_par);
      self -> Responsabile = DatabaseUser(); }

Ref<Documentazione> Documentazione::init
                              (string Nome_par)
{     (Ref<File>) self -> init(Nome_par);
      self -> Validità = EndOfPeriod; }
```

Si noti che il metodo **init**, definito nell'ambito della classe File, viene riutilizzato nell'ambito delle due sotto-classi; il nome del tipo `Ref<File>`, che compare racchiuso tra parentesi prima dell'invocazione di **init** all'interno dei metodi per Sorgente e Implementazione, rappresenta l'applicazione dell'operatore di *cast*, con il quale si specifica di considerare la variabile come istanza di un tipo diverso rispetto a quello dichiarato; in entrambi i casi il metodo **init** che viene richiamato è quello definito nell'ambito della classe File. Questo esempio ci mostra pertanto un'insieme di metodi che soddisfano l'ultimo criterio di buona progettazione introdotto nel Paragrafo 20.1.3, cioè il riutilizzo dei metodi definiti nelle classi generiche nell'implementazione dei metodi nelle classi più specifiche.

Si noti inoltre l'uso delle funzioni `DatabaseUser()` e `Today()`, e della variabile globale `EndOfPeriod`, che pur essendo oggetti accessibili a livello globale sono accettabili nel contesto delle tre implementazioni dei metodi. Tramite queste definizioni, è possibile inizializzare i file indipendentemente dalla loro appartenenza alla gerarchia; nel codice seguente, è necessario semplicemente porre al posto di Classe uno qualunque dei tre nomi di classi introdotti nell'esempio:

```
Ref<Classe> X;
X = new(Classe);
X -> init("MioProgramma");
```

20.1.7 Ridefinizione con raffinamenti di tipo

I meccanismi di ridefinizione visti nel paragrafo precedente non modificano l'interfaccia dei metodi. È però possibile raffinare attributi e metodi modificandone l'interfaccia, introducendo i meccanismi di sotto-tipazione.

La sotto-tipazione è una relazione fra tipi. Intuitivamente, T_1 è un sotto-tipo di T_2 se ammette un insieme di valori possibili più specifico. Per esempio, se T_2 è un tipo enumerativo, T_1 può essere definito come un sotto-insieme dei valori di T_2. Ogni tipo è sotto-tipo di se stesso, e nei sistemi in cui le classi sono interpretate anche come tipi, due classi in relazione di sotto-classe sono anche in relazione di sotto-tipo.

Un caso importante è quello dei record: dato un tipo record $T_X = [A_1 : T_1, \ldots, A_m : T_m]$, un altro tipo record T_Y è un sotto-tipo di T_X se ha la struttura $T_Y = [A_1 : T'_1, \ldots, A_m : T'_m, A_{m+1} : T'_{m+1}, \ldots, A_n : T'_n]$, con T'_i sotto-tipo di T_i per $1 \leq i \leq m$ e nessuna restrizione su T'_i per $(m+1) \leq i \leq n$; il sotto-tipo può cioè avere attributi T'_i che possono essere sotto-tipi degli attributi T_i, e avere attributi aggiuntivi che lo rendono più specifico. In realtà, abbiamo già visto che l'aggiunta di attributi caratterizza normalmente la definizione delle sotto-classi. Data una definizione precisa di sotto-tipo, una ridefinizione con raffinamento può far riferimento sia agli attributi sia ai metodi.

- Si consideri la definizione di una classe C_2 che eredita da C_1 e un generico attributo $A : T$ di C_1. La *co-varianza degli attributi* consente di associare all'attributo A, ridefinito in C_2, un sotto-tipo T' del tipo T.
- Si consideri la definizione di una classe C_2 che eredita da C_1 e un generico metodo m caratterizzato da un certo numero di parametri di ingresso, di tipo T_i, e da un

parametro di uscita, di tipo T. La *co-varianza del parametro di uscita* si ottiene dando al metodo m' ridefinito in C_2 un sotto-tipo T' di T. Per quanto riguarda i parametri di ingresso:

- se, per ogni parametro di ingresso di m, T_i' è un sotto-tipo di T_i, si ha la *co-varianza dei parametri di ingresso*;
- se, per ogni argomento di ingresso di m, T_i è un sotto-tipo di T_i', si ha la *contro-varianza dei parametri di ingresso*.

La co-varianza degli attributi e dei parametri di ingresso dei metodi, adottata nella maggior parte dei sistemi, è la nozione più intuitiva e utile nello specializzare attributi e metodi, ma non assicura la correttezza della sostituzione dai parametri formali a quelli attuali a tempo di compilazione. Ciò è illustrato dal seguente esempio.

Riprendiamo la situazione del Paragrafo 20.1.6 e introduciamo due nuove classi Utente e Programmatore, la cui struttura però non interessa:

```
add class Utente ....
add class Programmatore inherits Utente ....

add class File
type tuple(Nome: string,
           Creatore: Utente,
           Data: date)
method init(Nome_par: string,
            Utente_par: Utente):
          File is public

add class Sorgente inherits File
type tuple(Creatore: Programmatore)
method init(Nome_par: string,
            Utente_par: Programmatore): Sorgente
    is public
...
Ref<File> File::init
            (string Nome_par, Utente Utente_par)
{     self -> Nome = Nome_par;
      self -> Creatore = Utente_par;
      self -> Data = Today();
      return(self); }

Ref<Sorgente> Sorgente::init(string Nome_par,
                             Programmatore Utente_par)
{     (Ref<File>)self -> init(Nome_par, Utente_par);
      return(self); }
```

Osserviamo che l'attributo Creatore viene ridefinito nel contesto della classe Sorgente: il creatore non è un generico utente, bensì un programmatore. Questo è un esempio di *ridefinizione co-variante di attributi*, che può aver senso da un punto di vista applicativo in quanto si vuole imporre che solo i programmatori possano creare file sorgenti.

Il metodo init riceve in ingresso il parametro relativo all'utente, e si aspetta che venga correttamente invocato: l'inizializzazione di un generico File può ricevere come parametro attuale un generico utente, ma l'inizializzazione di un Sorgente deve ricevere come parametro attuale un programmatore. Questo è un esempio di *ridefinizione co-variante dei parametri di ingresso*. Si noti l'impossibilità di controllare a tempo di compilazione la correttezza dell'invocazione del metodo, se è possibile che un oggetto possa migrare dinamicamente dalla classe Utente alla classe Programmatore.

Infine, il parametro di uscita del metodo viene anch'esso ridefinito, poiché quando esso è invocato in una sotto-classe restituisce un tipo più specifico. Questo è un esempio di *ridefinizione co-variante del parametro di uscita*, che invece non pone problemi di correttezza dei tipi.

20.2 Basi di dati relazionali a oggetti

Le "basi di dati relazionali a oggetti" (*Object-Relational*, da cui ORDBMS) si caratterizzano per la scelta di un approccio evolutivo e non rivoluzionario rispetto alle basi di dati relazionali; questi sistemi, cioè, si pongono come obiettivo la realizzazione di estensioni compatibili della nozione classica di tabella, che consentono di esprimere la maggior parte dei concetti degli OODBMS. In questo paragrafo, mostriamo il linguaggio utilizzato per definire strutture negli ORDBMS, per differenza rispetto alle basi di dati a oggetti "pure" (OODBMS) fin qui illustrate. Come già osservato, la distanza tra questi due tipi di sistemi non è elevata, specie nel modello dei dati, in cui vengono definiti concetti molto simili. Introdurremo il modello dei dati per poi mostrare alcune caratteristiche del linguaggio di interrogazione, utilizzando come riferimento la versione SQL-3 dello standard SQL, che include come aspetto importante l'estensione a oggetti del modello relazionale.

20.2.1 Modello dei dati di SQL-3

Il modello dei dati degli ORDBMS è compatibile con il modello dei dati relazionale. Pertanto, nel modello ORDBMS è possibile definire tabelle, quali per esempio la tabella Persona, in cui compaiono alcuni dei vincoli di integrità SQL:

```
create table Persona(
      Nome varchar(30) not null,
      Residenza varchar(30),
      CodFisc char(16) primary key)
```

Tuttavia, l'approccio suggerito negli ORDBMS è quello di definire prima il tipo TipoPersona, in modo da renderlo riutilizzabile. L'uso di costruttori di tipo complessi – che estendono significativamente la nozione di dominio presente in SQL – è la prima differenza significativa rispetto alle basi di dati relazionali.

Tipi strutturati I tipi in SQL-3 si distinguono in *tipi distinti* e *tipi strutturati*; i primi corrispondono al meccanismo di definizione dei domini di SQL visto nel Capitolo 4 e consentono di definire un alias per i tipi predefiniti del modello dei dati SQL; i secondi si utilizzano essenzialmente per costruire le strutture delle tuple da inserire nelle tabelle. Pertanto, il medesimo effetto della definizione precedente della tabella Persona si ottiene dalle due seguenti definizioni:

```
create type TipoPersona as (
      Nome varchar(30),
      Residenza varchar(30),
      CodFisc char(16))
not final
ref from (CodFisc)

create table Persona of TipoPersona (
    ref is CodFisc derived)
```

Si può osservare come la definizione di TipoPersona termini con due nuovi costrutti. La clausola `not final` specifica che il tipo ammette la definizione di sotto-tipi; la sintassi

SQL-3 obbliga la specifica di questa clausola in ogni definizione di tipo. La clausola `ref from (CodFisc)` specifica invece che l'attributo CodFisc costituisce l'attributo da utilizzare per realizzare riferimenti a istanze del tipo, ovvero svolge il ruolo di identificatore per il tipo; la clausola `ref` è obbligatoria tutte le volte che si prevede, come in questo caso, di utilizzare il tipo strutturato per la definizione di tabelle. La tabella Persona viene quindi creata come un contenitore di istanze del tipo TipoPersona. La sintassi SQL-3 obbliga a specificare come vengono realizzati i riferimenti alle istanze della tabella; in questo caso la dichiarazione `ref is CodFisc derived` specifica, in modo ridondante, che l'identificatore viene derivato dal valore dell'attributo CodFisc. Una differenza della definizione di Persona tramite il tipo è che il tipo TipoPersona può essere utilizzato anche in altre tabelle. È possibile quindi definire:

```
create table Industriale of TipoPersona (
    ref is CodFisc derived)

create table Pilota of TipoPersona (
    ref is CodFisc derived)
```

Si noti la corrispondenza fra le nozioni di oggetto e classe nell'ambito degli OODBMS e di tupla e tabella nell'ambito degli ORDBMS; negli ORDBMS, i termini di oggetto e di tupla possono essere usati in modo intercambiabile. Come negli OODBMS, è possibile usare costruttori di tipo in modo ortogonale per costruire tipi arbitrariamente complessi. È possibile inoltre usare riferimenti da un tipo a un altro tipo, e quindi creare le premesse per condividere oggetti nella base di dati. Riprendiamo l'esempio illustrato nel Paragrafo 20.1 (con qualche semplificazione e qualche variante dovuta alle caratteristiche di SQL-3) e illustriamo la definizione dei corrispondenti tipi tupla, che includono TipoPersona già visto. Si noti l'uso del costrutto `multiset`[3] e `ref` (per denotare un riferimento da un tipo a un altro).

```
create type TipoStab as (
      Nome varchar(25),
      Citta varchar(7),
      Addetti integer)
not final

create type TipoCostr as (
     Nome varchar(25),
     Presidente ref(TipoPersona),
     Stabilimenti TipoStab multiset )
not final
ref is system generated

create type TipoPartiAuto as (
     Motore char(10),
     Ammortizzatore char(5))
not final

create type TipoAuto as (
     Targa char(7),
     Modello varchar(30),
     Produttore ref(TipoCostr),
     PartiMeccaniche TipoPartiAuto)
not final
ref is system generated
```

[3] Il costruttore di insiemi `setof` non è previsto in SQL-3; versioni successive dello standard lo potranno aggiungere. La prima versione di SQL-3, SQL:1999, presentava solo il costrutto `array`.

Si noti che i tipi TipoStab e TipoPartiAuto vengono usati nell'ambito dei tipi TipoCostr e TipoAuto senza però introdurre il costrutto `ref`, e quindi senza che vengano introdotti oggetti indipendenti. In tal modo, si possono costruire tuple che comprendono al loro interno sotto-tuple, garantendo un'arbitraria complessità strutturale. L'opzione `ref is system generated` specifica che è compito del sistema generare l'identificatore delle istanze del tipo. A questo punto, è possibile creare tabelle per i soli concetti Automobile e Costruttore, che si uniscono alle tabelle Presidente e Pilota, già create.

```
create table Automobile of TipoAuto
( ref is AutomobileId system generated )

create table Costruttore of TipoCostr
( ref is CostrId system generated,
  Presidente with options scope Industriale)
```

Si noti che nella definizione dei tipi tupla TipoAuto e TipoCostr gli attributi AutomobileId e CostrId non compaiono; i nuovi termini rappresentano in effetti gli identificatori delle tuple, presenti in modo implicito nella definizione del tipo. La clausola `ref is AutomobileId system generated` conferma che i valori dell'identificatore d'oggetto sono creati dal sistema. Grazie all'introduzione dei nomi espliciti AutomobileId e CostrId, i valori dell'identificatore possono essere utilizzati nelle interrogazioni come gli altri attributi.

Si può inoltre osservare l'uso del costrutto `with options` che consente di fornire maggiori dettagli e introdurre restrizioni su componenti del tipo quando esso viene utilizzato nell'ambito di una tabella. In questo caso il costrutto viene utilizzato per associare la clausola `scope` all'attributo Presidente, con la quale si pone il vincolo che i valori presenti nell'attributo Presidente debbano essere riferimenti alle tuple appartenenti alla tabella Industriale; se tale clausola fosse omessa, i valori dell'attributo Presidente potrebbero essere riferimenti a generici oggetti di tipo TipoPersona, presenti in una qualunque tabella che utilizza tale tipo.

Gerarchie In SQL-3 è possibile definire gerarchie di tipo e di tabelle. Le gerarchie di tipo estendono tipi precedentemente definiti aggiungendo nuove proprietà; per esempio è possibile costruire un tipo TipoAutoStorica aggiungendo a TipoAuto l'attributo AnnoCostr, come segue:

```
create type TipoAutoStorica
     under TipoAuto as (
     AnnoCostr integer)
not final
```

Le gerarchie sulle tabelle presentano una completa analogia con le gerarchie sulle classi viste nel Paragrafo 20.1. Quindi, le sotto-tabelle hanno per tipo un sotto-tipo delle tabelle da cui ereditano, e ogni oggetto (tupla) presente in una sotto-tabella appare come oggetto (tupla) nelle tabelle di livello gerarchicamente superiore. La definizione di AutomobileStorica come sotto-tabella di Automobile, che richiede comunque una clausola `under` nel contesto della creazione della sotto-tabella, può — dal punto di vista della costruzione del tipo — avvenire in due modi. È possibile far riferimento al sotto-tipo precedentemente definito, come segue:

```
create table AutomobileStorica of TipoAutoStorica
     under Automobile
```

In alternativa, è possibile definire in un unico passo tabella e tipo, con l'indicazione esplicita del tipo di AutomobileStorica, come segue:

```
create table AutomobileStorica of TipoAutoStorica
     under Automobile (AnnoCostr integer)
```

Si osservi come la sintassi SQL-3 richieda di non ripetere nella definizione di sotto-tabelle la clausola che esplicita la gestione dei riferimenti, in quanto anch'essa viene ereditata.

Metodi e funzioni I tipi definiti dall'utente in SQL-3 possono essere esplicitamente etichettati come tipi instanziabili o meno ([not] instantiable). I tipi classificati come non instanziabili non possono essere utilizzati direttamente nella definizione di tabelle; possono essere invece utilizzati come base per la definizione di sotto-tipi (in modo analogo alle classi *astratte* del modello a oggetti del linguaggio Java) o come componenti all'interno della definizione di altri tipi o tabelle. È inoltre possibile corredare i tipi, instanziabili o meno, di un insieme di metodi, che possono essere definiti in SQL-3 oppure in un linguaggio esterno di programmazione. I metodi di SQL-3 sono del tutto analoghi ai metodi discussi nel Paragrafo 20.1.3, e in particolare includono realizzazioni standard per il *costruttore*, gli *accessori* ai vari attributi e i *trasformatori*; è possibile negare agli utenti i privilegi d'accesso ai metodi, che vengono definiti come privilegi speciali, ottenendo l'effetto di incapsulare i dati.

Vediamo per esempio la definizione del tipo non instanziabile TipoPartiAuto, che comprende anche i metodi PotUnitaria e MaggiorPotUnitaria per esprimere, rispettivamente, il calcolo della potenza di ciascun cilindro del motore e un confronto tra l'istanza su cui viene invocato il metodo e un'altra istanza dello stesso tipo.

```
create type TipoPartiAuto as (
     Motore char(10),
     NumCilindri integer,
     Potenza integer,
     Cilindrata integer )
not instantiable
not final
method PotUnitaria ( )
  returns float,
method MaggiorPotUnitaria (
        ParteCfr TipoPartiAuto )
  returns boolean,
method DisegnoDisponibile ( )
  returns boolean
  language C
```

I metodi così definiti possono essere realizzati in SQL-3, oppure in un linguaggio di programmazione esterno; per esempio, il metodo DisegnoDisponibile è definito come esterno. In entrambi i casi, la specifica dei metodi richiede di definirne la *segnatura*, che, come nel Paragrafo 20.1.3, individua i parametri d'ingresso e i parametri d'uscita, e poi l'*implementazione*. Per la segnatura si usa una notazione funzionale, in cui i parametri d'ingresso sono racchiusi in parentesi; per ogni parametro di ingresso si definisce il nome e il tipo. Il tipo del parametro d'uscita, che è unico (e può essere omesso) viene indicato dopo la clausola returns.

```
create instance method PotUnitaria( )
  returns float
  for TipoPartiAuto
  return (cast (Self.Potenza as float) /
          Self.NumCilindri )
```

```
create instance method MaggiorPotUnitaria
                    (ParteCfr TipoPartiAuto)
  returns boolean
  for TipoPartiAuto
  return
   ((Self.PotUnitaria > ParteCfr.PotUnitaria)
    or((Self.PotUnitaria = ParteCfr.PotUnitaria)
        and (Self.Cilindrata > ParteCfr.Cilindrata)))
```

Qui sopra vediamo la definizione dei metodi PotUnitaria e MaggiorPotUnitaria, la cui segnatura è stata specificata nell'ambito della dichiarazione di TipoPartiAuto. Il metodo PotUnitaria restituisce la potenza erogata da ciascun cilindro; MaggiorPotUnitaria confronta invece l'istanza di TipoPartiAuto su cui viene invocato il metodo con un'altra istanza dello stesso tipo che viene passata come parametro. Le due implementazioni sono autoesplicative, in quanto si limitano a semplici computazioni e confronti sui valori degli attributi dei due parametri, raggiunti con una notazione a punto (*dot notation*) per estrarre l'attributo di un oggetto referenziato da una variabile. È da notare come in MaggiorPotUnitaria venga utilizzato il metodo PotUnitaria; si osserva tra l'altro che la sintassi di SQL-3 non impone di usare una coppia di parentesi dopo il nome del metodo, come viene invece fatto in molti sistemi a oggetti per distinguere i metodi dagli attributi statici.

20.2.2 Linguaggio di interrogazione di SQL-3

SQL-3 è pienamente compatibile "all'indietro" con SQL-2, ovvero ogni comando corretto per la sintassi SQL-2 continua a essere un comando valido in SQL-3. Pertanto, in SQL-3 è possibile definire query relazionali "standard" sulle tabelle "standard". Per esempio:

```
select Nome
from Persona
where CodFisc = 'TRE SFN 56D23 S541S'
```

La navigazione lungo i riferimenti fra tipi richiede in SQL-3 l'operatore di dereferenziamento (*dereferencing*); tale operatore consente di accedere da un oggetto sorgente x a un attributo A di un oggetto y referenziato in x, nel modo seguente: `x -> A`. Il seguente esempio mostra l'uso dell'operatore di dereferenziamento per accedere al valore dell'attributo Nome dell'oggetto della tabella Industriale "puntato" dall'oggetto della tabella Costruttore che è "puntato" a sua volta dall'automobile che soddisfa il predicato `Targa = 'DB123MS'`.

```
select Produttore -> Presidente -> Nome
from Automobile
where Targa = 'DB123MS'
```

In SQL-3 gli attributi di tipo OID possono essere esplicitamente utilizzati nelle interrogazioni, e in particolare possono essere confrontati tramite l'operatore di uguaglianza con i riferimenti a tuple dello stesso tipo. L'ultima interrogazione può essere perciò espressa anche come segue:

```
select Industriale.Nome
from Automobile, Costruttore, Industriale
where Automobile.Targa = 'DB123MS'
  and Automobile.Produttore = Costruttore.CostrId
  and Costruttore.Presidente = Industriale.CodFisc
```

L'interrogazione costruisce un join fra le tabelle Automobile, Costruttore e Industriale, in cui l'attributo **Produttore** di Automobile viene confrontato con l'identificatore di Costruttore e l'attributo **Presidente** di Costruttore viene confrontato con l'identificatore di Industriale.

20.3 Estensioni tecnologiche per le basi di dati a oggetti

Le basi di dati a oggetti utilizzano la tecnologia dei sistemi di gestione di basi di dati, descritta nei capitoli precedenti, ma introducono alcune estensioni tecnologiche abbastanza significative.

20.3.1 Rappresentazione dei dati e degli identificatori

Un primo problema che si incontra nella gestione degli oggetti è posto dalla rappresentazione di oggetti complessi in memoria di massa, cioè nei cosiddetti *object server*; per memorizzare una gerarchia di classi tramite file di memoria di massa esistono due approcci estremi.

- L'approccio *orizzontale* consiste nel memorizzare ogni oggetto in modo contiguo; in particolare, tutte le istanze di una stessa classe vengono memorizzate all'interno dello stesso file; si avrebbe pertanto, relativamente all'esempio descritto in Figura 20.2, un file ciascuno per le classi Automobile, AutoSportiva, AutoStorica e AutoSportivaStorica. Con questo approccio, l'accesso a un singolo oggetto risulta particolarmente efficiente, mentre risulta gravoso l'accesso in base a specifiche proprietà degli oggetti (per esempio, l'accesso a una generica automobile in base all'attributo **Colore** richiede l'accesso a quattro file).
- L'approccio *verticale* consiste nel memorizzare contiguamente le medesime proprietà, spezzando un oggetto nelle sue componenti. Questo approccio è anche detto "normalizzato" in quanto potrebbe essere direttamente applicato per il modello relazionale. Abbiamo anche in questo caso un file ciascuno per le quattro classi citate, però per esempio il file *Automobile* contiene l'informazione relativa al **Modello** di tutti gli oggetti che sono istanza o membri della classe Automobile; gli oggetti vengono ricostruiti a partire da riferimenti fra i loro componenti, e l'accesso a tutti gli attributi di un oggetto della classe AutoSportivaStorica richiede l'accesso a quattro file.

Queste soluzioni ripercorrono le varie alternative di traduzioni dal modello Entità-Relazione al modello relazionale che caratterizza il progetto logico, visto nel Capitolo 8. La soluzione orizzontale è più coerente con il paradigma a oggetti in quanto gestisce ciascun oggetto in modo unitario, però pone problemi quando un oggetto può essere istanza di più classi. In generale, qualunque soluzione intermedia è accettabile e può essere suggerita dalle esigenze di una particolare applicazione.

La soluzione verticale si adatta anche alla gestione della complessità strutturale (dovuta all'uso dei costruttori di tipo); in questo caso però i riferimenti tra sotto-oggetti devono essere creati all'atto di partizionare un oggetto in parti e non possono utilizzare gli identificatori assegnati agli oggetti. Inoltre, contatori e identificatori opportunamente generati dal sistema sono necessari per gestire insiemi, liste e multi-insiemi.

Negli OODBMS specializzati nella gestione di documenti e dati multimediali, questi vengono tipicamente rappresentati come oggetti binari (i cosiddetti *binary long objects*, o *blob*) e memorizzati su file specifici (uno per oggetto binario).

Un problema caratteristico degli OODBMS è la rappresentazione degli OID, per i quali sono date due soluzioni:

- utilizzare un *indirizzo fisico*, cioè che include l'allocazione fisica (blocco) dell'oggetto in memoria di massa. L'ovvio vantaggio è la rapidità di accesso, lo svantaggio è la criticità della riallocazione di un oggetto, che può essere gestita tramite un indirizzamento indiretto, lasciando cioè nella pagina fisica in cui l'oggetto è inizialmente creato un "puntatore" alla sua posizione corrente;
- utilizzare un *surrogato*, cioè un valore che venga associato univocamente a un oggetto tramite un algoritmo (per esempio, in un sistema distribuito, tramite un contatore di oggetti per ogni nodo del sistema); in genere si predispone poi un indice oppure un meccanismo di hashing che consenta di passare dal surrogato all'indirizzo fisico di ciascun oggetto, in modo da garantire il loro reperimento efficiente. È possibile, anche se meno frequente, utilizzare surrogati tipizzati associati alle varie classi; è necessario in tal caso disporre di distinti contatori per ogni classe.

Gli OID possono essere di dimensioni elevate: in molte implementazioni di sistemi distribuiti raggiungono le dimensioni di 64 byte.

20.3.2 Indici complessi

Nei sistemi a oggetti è importante consentire accessi efficienti lungo i cammini (*path expression*), in modo da consentire l'esecuzione efficiente di query e programmi; per questo scopo vengono sviluppati negli OODBMS gli *indici complessi*.

Supponiamo che, nella base di dati a oggetti rappresentata in Figura 20.2, la nuova classe Città sia raggiungibile dalla classe Stabilimento. Consideriamo un cammino complesso che parta da un oggetto x appartenente alla classe Automobile:

```
x.Produttore.Stabilimenti.Città.Nome
```

Per eseguire una query o un programma che utilizza questo cammino, occorre passare dalle automobili alle città a esse collegate, le quali avranno sull'attributo Nome valori di tipo stringa, quali per esempio "Milano" o "Torino". In questo modo si ritrovano gli oggetti di Automobile che soddisfano il predicato:

```
X.Produttore.Stabilimenti.Città.Nome = "Torino"
```

Gli indici complessi che possono essere definiti in un OODBMS sono di tipo assai diverso.

- Un'organizzazione *multi-indice* garantisce la presenza di un indice per ciascuna proprietà utilizzata lungo il cammino. Quindi, procedendo a ritroso:
  - un indice dalle costanti di tipo stringa che contengono i nomi di città agli oggetti Città;
  - un indice dagli oggetti Città agli oggetti Stabilimento;
  - un indice dagli oggetti Stabilimento agli oggetti Costruttore;
  - un indice dagli oggetti Costruttore agli oggetti Automobile.

  Usando i tre indici in un modo che ricorda il metodo *nested-loop* per l'esecuzione dei join (Paragrafo 11.6.2), si ritrovano tutti gli oggetti della classe Costruttore che soddisfano la condizione.
- Un *indice nidificato* consente il passaggio diretto dai valori che compaiono al termine del cammino alla classe che è all'origine del cammino. Dunque, dalla costante "Torino" si ricavano immediatamente tutti gli oggetti della classe Costruttore che soddisfano la condizione.
- Infine, un *indice cammino* consente il passaggio dai valori che compaiono al termine del cammino agli oggetti di tutte le classi che compaiono lungo il cammino.

Gli indici così introdotti vengono realizzati con alberi B+ di struttura particolare. Occorre tener conto nella loro gestione delle creazioni e cancellazioni di oggetti intermedi, che possono modificare in modo non banale la struttura degli indici; tecniche abbastanza sofisticate gestiscono al meglio queste operazioni.

20.3.3 Architettura client-server

Anche negli OODBMS gli oggetti vengono memorizzati in server dedicati alla gestione dei dati; poiché le applicazioni delle basi di dati a oggetti sono più recenti, l'architettura client-server è ancora più diffusa che non in un contesto relazionale. Cambia però il paradigma di interazione con la base di dati, che nella maggioranza dei casi avviene tramite programmi applicativi scritti in un linguaggio tradizionale. Perciò negli OODBMS la programmazione imperativa si sostituisce a un linguaggio associativo di interrogazione, quale è SQL, che nel Paragrafo 15.1 è indicato come uno dei principali motivi del successo delle architetture client-server nel contesto dei sistemi relazionali.

Questo cambio di paradigma porta a una nuova suddivisione di compiti nella architettura client-server, che sta emergendo nei sistemi a oggetti, in cui il sistema client *importa* interi oggetti; in questo modo, i programmi applicativi possono venire eseguiti direttamente nei buffer dei sistemi client.

Si pensi per esempio a un'applicazione di tipo ingegneristico, in cui l'utente progetta un componente meccanico oppure un chip: una volta che l'intero oggetto è caricato sulla sua workstation, l'utente non interagisce più con l'object server per tutta la durata di una sessione di progettazione. Se viceversa l'oggetto restasse nel buffer del server durante la sessione di progettazione, si avrebbero pesanti interazioni tra client e server, con sovraccarico della rete locale.

Per facilitare l'import/export di oggetti, molti sistemi utilizzano in memoria centrale la stessa rappresentazione degli oggetti che esiste in memoria di massa, e quindi trasferiscono dai server ai client intere pagine di memoria. Una tipica ottimizzazione che avviene all'atto del caricamento degli oggetti dalla memoria di massa ai buffer di memoria centrale è la conversione degli OID; questa operazione prende il nome di *pointer swizzling*, ed è giustificata dalla maggiore efficienza e compattezza degli indirizzi di memoria centrale rispetto a quelli di memoria di massa.

Alcuni sistemi mirano esclusivamente a ottimizzare la velocità di accesso, e in tal caso il puntatore viene *riscritto* esattamente al di sopra dell'OID, lasciando la struttura della pagina inalterata. Per ottimizzare il processo di conversione, il passaggio da OID a puntatore può essere posposto al momento in cui un'applicazione utilizza realmente il puntatore (in quanto, in base al principio di località dei dati, ci sono buone probabilità che il riferimento verrà usato anche successivamente). All'atto di ricaricare gli oggetti in memoria di massa è necessario operare la conversione inversa; apposite strutture dati mantengono la corrispondenza fra puntatori in memoria centrale e OID.

20.3.4 Transazioni

Sul fronte transazionale, gli OODBMS offrono, accanto alle classiche proprietà acide delle transazioni, anche meccanismi meno restrittivi, che si basano sull'ipotesi di una possibile cooperazione fra gli utenti degli OODB. Vediamo brevemente i requisiti di alcuni di questi meccanismi.

- Nel controllo di concorrenza, vengono tipicamente definite delle operazioni di *check-out* degli oggetti che consentono di caricare interi oggetti da un server OODBMS alla workstation su cui opera un utente, per manipolarli durante lunghe sessioni di lavoro, consentendo però ad altri utenti di accedere ancora all'oggetto presente sul server. Ovviamente, l'operazione duale di *check-in*, in cui l'oggetto viene riportato sul server, richiede la cooperazione con gli altri progettisti.
- Un altro modo di garantire processi concorrenti cooperativi è basato sull'uso di *versioni*, cioè di varie copie dello stesso oggetto a istanti progressivi. In tal caso, è anche possibile definire relazioni fra oggetti che rappresentino l'allineamento fra differenti versioni. Questo problema è particolarmente critico nel contesto

degli ambienti CASE (*Computer-Aided Software Engineering*), in cui si vuole consentire e documentare l'evoluzione dei programmi e garantire che varie versioni allineate costituiscano un'unica unità logica (per esempio, rispetto al processo di compilazione e link).

- In alcune applicazioni possono essere previste *transazioni di lunga durata* (*long transaction*); un esempio è la transazione che descrive il processo di check-out e check-in di un oggetto, che può essere trasferito su una workstation di progettazione per intere giornate.
- Infine, le transazioni possono essere composte da *transazioni complesse* (*nested transaction*); questo approccio, presente anche nei sistemi relazionali, si verifica per esempio quando un processo client deve realizzare un compito unitario colloquiando con vari server distinti che non consentono l'uso di transazioni globali; in tal caso, l'atomicità globale viene ottenuta tramite un coordinamento fra le transazioni acide di basso livello. L'esempio più tipico è l'organizzazione di un viaggio, che richiede di accedere a un sistema di prenotazioni alberghiere, una compagnia di auto-noleggi e ai sistemi di prenotazione di varie compagnie aeree. Ciascuna operazione di inserzione o cancellazione di una prenotazione su di uno dei DBMS è gestita come una transazione acida. Quando però l'impossibilità di prenotare una delle risorse viene rilevata dopo averne prenotata un'altra, l'annullamento della prenotazione già effettuata avviene tramite una *transazione di compensazione*, attivata dal coordinatore della transazione complessa. L'atomicità globale viene garantita, in modo asintotico, dal gestore delle transazioni complesse.

20.4 Basi di dati multimediali

Gli ultimi anni hanno visto crescere l'esigenza di gestire, accanto ai dati alfanumerici, altri dati che rappresentano documenti, immagini, video, audio, che indichiamo genericamente come "dati multimediali"; con il termine "base di dati multimediale" intendiamo quindi un sistema dotato della capacità di memorizzare, interrogare e presentare dati multimediali. È ragionevole discutere le problematiche delle basi di dati multimediali in questo capitolo in quanto è possibile vedere i dati multimediali come particolari tipi di dati, per i quali sono necessarie astrazioni specifiche che ne consentano una gestione efficace. Tuttavia, la capacità di gestire dati multimediali può essere offerta anche dalle basi di dati relazionali.

20.4.1 Tipi di dati multimediali

Iniziamo ad analizzare le caratteristiche dei principali dati multimediali: immagini, audio, video, documenti e annotazioni.

Immagini L'esigenza di associare a un oggetto, spesso descritto tramite attributi alfanumerici, anche una sua immagine è sempre più diffusa. Basi di dati di immagini sono utilizzate in campo clinico, ove per esempio associano a ciascun paziente le sue radiografie; le polizie di tutto il mondo si scambiano schede segnaletiche dei ricercati, in cui compaiono le immagini del loro volto; le agenzie immobiliari illustrano le case in vendita tramite immagini fotografiche; la descrizione delle località turistiche avviene soprattutto attraverso immagini. La principale difficoltà nella gestione delle immagini è il numero elevato di bit necessari per memorizzarle in formato binario; per ridurre tale numero si usano formati standard (tra cui: GIF, JPEG, TIFF, PNG), che ne consentono la rappresentazione in forma compressa.

Audio I dati audio possono contenere conversazioni, musiche e altri segnali sonori (per esempio, i suoni che vengono generati quando le applicazioni vogliono segnalare

degli eventi). Un segnale audio viene rappresentato mediante una serie di valori numerici che rappresentano l'ampiezza del segnale sonoro in una finestra temporale di piccole dimensioni. Per esempio, nella modalità di campionamento usata nei CD audio, si ha un'osservazione ogni 20 microsecondi rappresentata con un valore a 16 bit. Applicazioni che non richiedono la massima fedeltà di riproduzione dei suoni possono usare una frequenza di campionamento più bassa e meno bit per la rappresentazione del valore. Nella maggior parte delle applicazioni si utilizzano delle tecniche di compressione per limitare la dimensione di questi dati, che possono altrimenti diventare molto voluminosi. Le tecniche di compressione possono essere senza perdita di informazioni (*lossless*), o possono comportare una leggera perdita di informazione (*lossy*), tipicamente con un impatto non percettibile da parte di chi ascolta nella qualità del segnale e un guadagno molto significativo in termini di riduzione della dimensione dei dati.

Video I video sono collezioni di immagini (o *frame*) mostrate l'una dopo l'altra da un dispositivo riproduttore. Un video può illustrare un evento storico, una lezione, gli animali in azione nel loro habitat naturale e così via. Se la memorizzazione di immagini pone problemi di spazio, questi sono assai più significativi nel caso di video; basti pensare che un video di 60 minuti può contenere più di 100.000 frame. Gli standard più diffusi per la gestione dei video (MPEG-1, MPEG-2, MPEG-4) consentono qualità diverse ottenute con diversi livelli di compressione; per esempio, MPEG-1 non ha sufficiente qualità per garantire la riproduzione televisiva, MPEG-2 consente tale riproduzione aumentando la qualità del video (ma richiede anche un maggior numero di bit) e MPEG-4, lo standard oggi maggiormente usato, migliora ulteriormente la qualità consentendo anche la riproduzione televisiva ad alta definizione.

Documenti I documenti sono composti da testi e immagini, presentati tramite un preciso formato. Per esempio, la pagina di un giornale, oppure la home page degli autori di questo libro, oppure una lettera d'affari scritta su carta intestata e firmata dall'autore sono esempi di documenti. Le cosiddette "biblioteche digitali" si ripromettono di memorizzare e rendere disponibili via Internet milioni di libri e altri documenti. Per la costruzione di documenti si possono usare linguaggi di markup, quali HTML o XML (si veda il Capitolo 13); questi linguaggi sono in grado di includere in uno spazio bidimensionale immagini e testi, composti secondo il formato prescelto dall'autore del documento (per esempio, per quanto riguarda i testi, con la scelta dei caratteri, l'uso di grassetto e sottolineature, allineamenti e indentazioni ecc.).

Annotazioni Infine, le annotazioni sono testi liberi, che vengono aggiunti ad altri dati multimediali (immagini, documenti, suoni). Le annotazioni consentono di corredare un generico dato multimediale con indicazioni specifiche (per esempio, il proprio gradimento, o l'esigenza di collegare un documento a un altro). Dato che ciascuna annotazione ha carattere personale, nella gestione delle annotazioni assume un ruolo fondamentale l'utente, che può per esempio decidere di arricchire un documento con molte annotazioni personali, stabilendo poi che alcune di esse siano condivisibili – cioè accedibili opzionalmente da altri utenti – e altre siano invece private.

20.4.2 Interrogazioni su dati multimediali

Mentre la codifica dei dati multimediali è un problema generale della multimedialità, l'abilità di interrogare grandi moli di dati multimediali per estrarre dati di interesse è un problema più specifico delle basi di dati multimediali. In questo paragrafo, ci soffermiamo su alcuni classici esempi di interrogazioni che si applicano ai dati

multimediali, occupandoci più della loro formulazione che non della descrizione delle tecniche per calcolarne il risultato.
Per esempio, un'interrogazione a un archivio di immagini può richiedere l'estrazione di immagini con certi tratti particolari: tutte le radiografie polmonari in cui compaiono segni di broncopolmonite, oppure l'individuo più simile all'identikit trasmesso dalla polizia e relativo a un rapinatore di banche, o ancora tutti i ritratti rinascimentali di madonne con bambino conservati al museo del Louvre. Se già la ricerca di un'immagine in base alle sue caratteristiche appare difficile, ancor più difficile appare il compito di estrarre segnali audio oppure video in base a pattern particolari (tutte le musiche di Wagner in cui compare un particolare tema, oppure le sequenze video in cui compaiono goal della nazionale di calcio brasiliana), in quanto in tal caso il pattern deve essere riconosciuto operando su molteplici frame o finestre temporali. L'interrogazione dovrà di norma sfruttare inizialmente informazioni di tipo strutturale, associate ai dati multimediali, per estrarre un sotto-insieme di essi; dopodiché sarà possibile valutare, per ciascun dato multimediale estratto, la "somiglianza" rispetto alla richiesta formulata. In genere, si potrà selezionare un dato multimediale solo in termini probabilistici, estraendo cioè quei dati che soddisfano la richiesta con una probabilità data. Sarà quindi possibile presentare il risultato di un'interrogazione partendo dai dati che hanno maggior probabilità di soddisfare la richiesta, e arrestandosi al di sotto di una certa soglia di probabilità. Tecniche adattative consentono agli utenti di guidare la ricerca, associando ai dati estratti un gradimento che permette di indirizzare la ricerca in modo che evolva in base all'esito dell'interazione con l'utente.

20.4.3 Ricerca di documenti

Il caso di gran lunga più ricorrente di interrogazione su dati multimediali è l'estrazione di documenti che contengono particolari informazioni testuali. Si possono in tal caso utilizzare tecniche ormai collaudate e di grande successo, che fanno riferimento alla disciplina del *ritrovamento delle informazioni* (*Information Retrieval*) su testi; per rendersi conto dell'efficacia di tali metodi, basti pensare alla qualità dei cosiddetti "motori di ricerca", che consentono di estrarre i siti web di interesse a partire da poche parole chiave.

Queste tecniche si basano innanzitutto sulla capacità di estrarre da un testo informazioni utili per poter decidere se esso sia rilevante nell'ambito di una query. Tali informazioni si riducono a una rappresentazione ottimizzata delle principali parole chiave presenti nel testo, con associata un'indicazione relativa alla loro frequenza di occorrenza. Per costruire queste informazioni relativamente a un generico testo è necessario operare come segue:

- escludere dal testo parole irrilevanti (articoli, congiunzioni, preposizioni ecc.), che compaiono con elevata frequenza ma non sono significative;
- ridurre parole simili a un'unica parola chiave. Per esempio, le parole "abita", "abitazione", "abitato", "abitante" sono tutte legate a un unico concetto di "abitare" e quindi sostituibili con quest'ultima parola;
- associare a ogni parola chiave così definita la propria frequenza, definita come il numero di occorrenze della parola rispetto al numero totale di parole presenti nel testo.

A questo punto, la ricerca di un testo che soddisfi l'interrogazione di un utente, espressa tramite espressioni booleane di parole chiave, si riduce a determinare quel testo in cui le parole chiave proposte dall'utente, in una combinazione compatibile con quella richiesta dall'utente, hanno frequenza più elevata. Per meglio definire l'efficacia della ricerca, si definiscono due misure: la *precisione* (*precision*) e il *richiamo* (*recall*). Supponiamo che per ogni documento sia noto, in modo deterministico, se esso è rilevante, cioè se fa parte o meno del risultato dell'interrogazione.

- La precisione indica la percentuale dei documenti rilevanti all'interno dei documenti estratti.
- Il richiamo indica invece la percentuale dei documenti rilevanti estratti rispetto al totale dei documenti rilevanti nella base di dati.

Un buon algoritmo di ricerca deve cercare di tener conto di entrambi questi fattori: deve offrire una buona precisione, in modo da presentare documenti che sono in larga misura rilevanti, ma anche un buon richiamo, in modo da ridurre il rischio di omettere dalla risposta i documenti più rilevanti per l'utente.

Varie tecniche sono disponibili per organizzare l'informazione che descrive un testo, comprimendola e strutturandola in modo tale da rendere più efficace la ricerca. In particolare, è possibile rappresentare più documenti tramite *matrici*, in cui righe e colonne di una matrice rappresentano rispettivamente parole chiave e identificatori dei documenti, e ciascun elemento della matrice rappresenta la frequenza della parola chiave all'interno del documento. In tal caso, l'estrazione dei documenti e la compressione dei dati a essi relativi sono affidate a tecniche di calcolo matriciale.

In alternativa è possibile rappresentare la corrispondenza fra parole chiave e documenti che le contengono tramite *indici invertiti*; gli indici sono organizzati sulle parole chiave, come illustrato nel Capitolo 11, e al loro interno, come foglie, vi sono gli identificatori dei documenti che contengono tali parole chiave. Tramite queste strutture dati è facile estrarre documenti che soddisfano condizioni booleane sulle parole chiave: per esempio l'intersezione di due parole chiave si ottiene semplicemente facendo l'intersezione degli insiemi di identificatori ottenuti percorrendo l'indice due volte, con le due parole scelte dall'utente.

Altre strutture dati mettono in evidenza per prime le parole chiave a elevata frequenza; per esempio, la *segnatura* di un documento è una rappresentazione compatta delle n parole chiave che compaiono in un documento con frequenza più elevata; tecniche sofisticate consentono di estrarre documenti oppure di decidere se due documenti sono simili in base alla loro segnatura.

20.4.4 Rappresentazione dei dati spaziali

I dati spaziali descrivono l'informazione presente in uno spazio a n dimensioni, per esempio una mappa geografica (bi-dimensionale) o il progetto di un edificio in costruzione (tri-dimensionale). La gestione dei dati spaziali è un ambito applicativo abbastanza specifico che ha però acquisito grande importanza; per questo motivo, essa viene spesso affidata a sistemi dedicati, tra cui presentano la relazione più stretta con il mondo delle basi di dati i cosiddetti *sistemi informativi geografici* (*Geographic Information Systems*, GIS).

Il problema principale della gestione dei dati spaziali è la scelta di una struttura dati che consenta di rispondere a interrogazioni relative alla disposizione dei dati nello spazio. Per esempio: estrarre tutti i punti di una mappa che distano meno di una determinata distanza da un punto particolare; oppure determinare tutte le regioni che sono adiacenti a una regione data; o ancora, determinare tutti i punti di una mappa che si trovano lungo una linea, e potrebbero rappresentare le città collocate lungo il corso di un fiume. Ovviamente, i GIS sono in grado di descrivere non solo la disposizione dei dati nello spazio, ma anche le caratteristiche di ciascun punto, linea, o regione dello spazio; per esempio, i punti che descrivono le *città* saranno dotati, oltre alle proprie coordinate geografiche, anche di altre informazioni, quali per esempio il numero di abitanti e l'altezza sul livello del mare; le regioni saranno invece caratterizzate, per esempio, dal tipo di cultura agricola prevalente, oppure dalla quantità media mensile di pioggia per unità di superficie. Le linee potranno rappresentare fiumi, oppure strade o ferrovie. Tramite i GIS sarà allora possibile esprimere interrogazioni in cui si mescolano aspetti spaziali e non; per esempio, si potranno estrarre tutte le

regioni italiane in cui si produce il vino Chianti, oppure le città lombarde con meno di centomila abitanti in cui si trova un castello medioevale.
La gestione efficiente dei dati spaziali richiede di organizzare l'informazione tramite strutture dati speciali, in modo che la contiguità geografica dei dati sia rappresentata nella struttura dati, e che ciò consenta un'esecuzione efficiente delle interrogazioni che sfruttano tale contiguità. In particolare, vari tipi di strutture ad albero consentono di gestire collezioni di punti (per esempio, le città in una mappa geografica) e di rispondere efficientemente a interrogazioni del tipo: estrarre tutte le città che distano meno di una certa distanza da un determinato punto. Ciascun punto viene rappresentato da un nodo dell'albero; ogni nodo comprende le sue coordinate X e Y, un'informazione specifica del nodo e i puntatori ai nodi successori.

Nell'organizzazione ad *albero bidimensionale* (*2-d tree*) ogni nodo ha due successori, rispettivamente destro e sinistro; in questa organizzazione, il nodo radice rappresenta un'intera zona geografica, e ogni nodo suddivide la zona geografica da esso rappresentata in due zone tramite una linea, che è orizzontale oppure verticale a seconda che il punto sia a una distanza pari o dispari rispetto alla radice. Nella Figura 20.4, *A* è il nodo radice, *B* e *C* i suoi discendenti, *E* il discendente di *C*, *F* il discendente di *E* e *D* il discendente di *B*. Analizzando la figura si vede che *A* suddivide la figura verticalmente, *B* orizzontalmente, *C* orizzontalmente, *D* verticalmente, *E* verticalmente e *F* orizzontalmente.

Nell'organizzazione a *quad-tree* ogni nodo suddivide la zona geografica da esso rappresentata in quattro zone tramite due linee – orizzontale e verticale – che passano attraverso il punto stesso; quindi, ogni nodo ha quattro successori, che rappresentano i quattro quadranti. Nella Figura 20.5, *A* è il nodo radice, *B*, *C*, *D* e *E* i suoi discendenti, *F* il discendente di *E*. Ogni punto suddivide la regione in cui è collocato in quattro parti.

Vari sistemi commerciali sono dedicati specificamente alla gestione di dati spaziali; tra essi, ArcGIS, dotato di un sottosistema per la gestione di dati spaziali che si integra con i principali DBMS relazionali. L'Open Geospatial Consortium (precedentemente chiamato Open GIS Consortium) ha sviluppato una serie di standard relativi ai formati per lo scambio di dati spaziali, in modo da consentire interoperabilità fra i vari prodotti GIS.

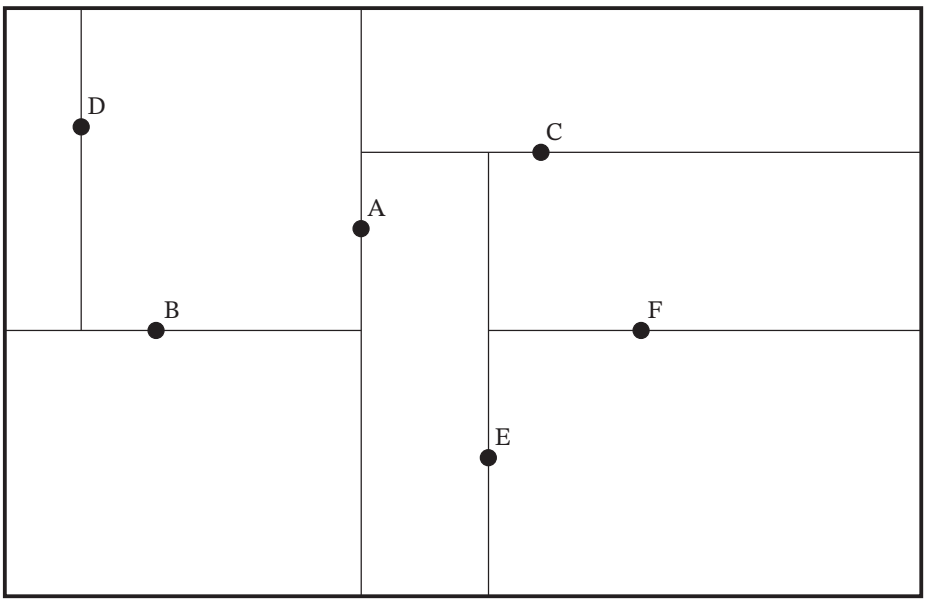

Figura 20.4
Un *2-d tree*.

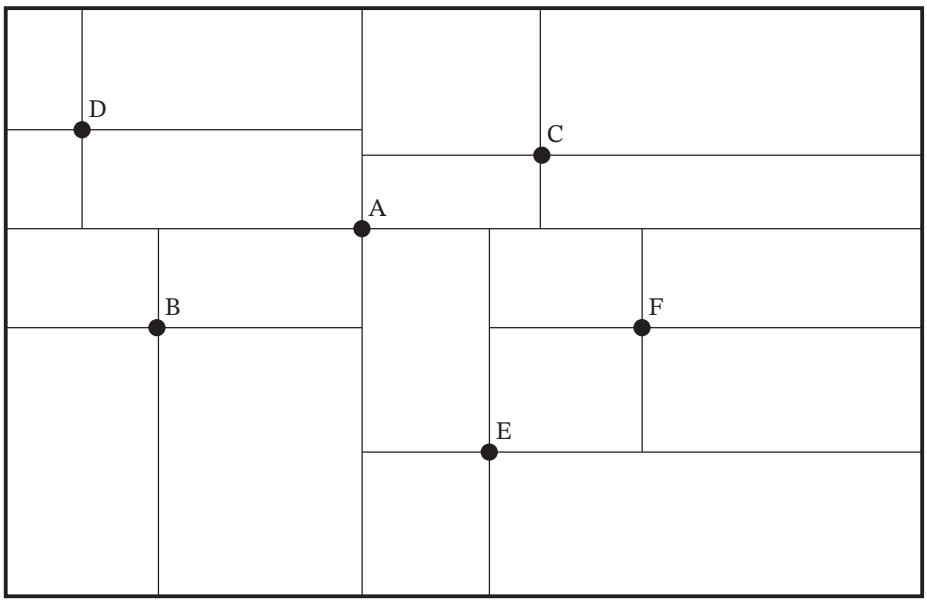

Figura 20.5
Un *quad-tree*.

Note bibliografiche

Vari libri di testo sono dedicati specificamente alle basi di dati a oggetti, tra cui il libro di Bertino e Martino *Sistemi di basi di dati orientati agli oggetti: concetti e architetture*, di Addison-Wesley, 1992 [100]. Altri libri su questo argomento sono: *Object Data Management - Object-Oriented and Extended Relational Database Systems*, Revised edition. Addison-Wesley, 1994 di Cattel [106] e *Object Databases: The Essentials* Addison-Wesley, 1995 di Loomis [136]. Il libro *The Story of* O_2, Morgan Kaufmann, 1992, curato da Bancilhon *et al.* [97], contiene vari articoli sul sistema O_2; il primo articolo riporta il cosiddetto "Manifesto delle basi di dati a oggetti". Una panoramica sulle basi di dati relazionali a oggetti è offerta da *Object-Relational DBMSs - The Next Great Wave*, Morgan Kaufmann, 1994 di Stonebraker [149], mentre il "Manifesto sulle basi di dati della terza generazione" è pubblicato in Third-Generation Database System Manifesto - The Committee for Advanced DBMS Function. *ACM SIGMOD Record*, vol. 19, n. 3, McGraw Hill, pagg. 31-44, 1990 di Stonebraker *et al.* [152]. La parte a oggetti dello standard SQL-3 è descritta nel testo di Melton *Advanced SQL: 1999*, Morgan Kaufmann, San Francisco, 2003 [140]. Le basi di dati multimediali sono descritte nel libro di Subrahmanian *Principles of Multimedia Database Systems*, Morgan Kaufmann, 1998 [153] , mentre un testo classico sulle strutture dati spaziali è il testo di Samet *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, 1989[145]. Discussioni e confronti con i sistemi SQL sono disponibili su molti siti Web e in alcuni articoli, fra cui quelli di Cattell Scalable SQL and NoSQL data stores. *SIGMOD Record*, vol. 39, n. 12, pagg. 12-27, 2010 [107], Cattell e Stonebraker 10 rules for scalable performance in 'simple operation' datastores. *Commun. ACM*, vol. 54, n. 6, pagg. 72-80, 2011 [150], e Atzeni *et al.* The relational model is dead, SQL is dead, and I don't feel so good myself. *SIGMOD Record*, vol. 42, n. 2, pagg. 64-68, 2013 [96].

Esercizi

20.1 Definire una classe FiguraGeometrica e tre sotto-classi che ereditano da essa Quadrato, Cerchio e Rettangolo. Definire un metodo `Area` la cui implementazione in FiguraGeometrica ritorna il valore zero, mentre nelle sotto-classi viene valutata in funzione delle proprietà delle sotto-classi medesime. Mostrare l'invocazione del metodo da parte di un programma che scandisce una lista di figure geometriche di tipo arbitrario.

20.2 Definire il dizionario dati di una base di dati a oggetti. Si suggerisce di introdurre classi e gerarchie di classi relative ai vari concetti di uno schema a oggetti: le classi, i tipi atomici, i tipi strutturati tramite i vari costruttori, le gerarchie di generalizzazione, i metodi con i loro parametri di ingresso e di uscita. Popolare il dizionario dati con dati che descrivono parte dello schema relativo alla gestione delle automobili, descritto in Figura 20.2. Pensare poi a qualche interrogazione che consenta di estrarre, per esempio, l'elenco delle classi e dei metodi con ridefinizione co-variante del tipo di uscita.

20.3 Si consideri il seguente schema di una base di dati a oggetti:

```
add class Città
  type tuple(Nome: string,
             Nazione: string,
             Monumenti: set(Monumenti),
             Hotels: list(Hotel));

add class Hotel
  type tuple(Nome: string,
             Indirizzo: tuple(Via: string,
                              Città: Città,
                              Numero: integer,
                              CAP: string);
             Stellette: integer,
             Attrattive: list(string));

add class Luogo
  type tuple(Nome: string,
             Fotografia: Bitmap,
             Indirizzo: tuple(Via: string,
                              Città: Città,
                              Numero: integer,
                              CAP: string);
             Attrazioni:
                  set(ServiziTuristici));

add class Monumento inherits Luogo
  type tuple(DataCostruzione: date,
             GiorniChiusura: list(string),
             PrezzoAmmissione: integer,
             Architetto: Person);

add class ServiziTuristici
  type tuple(Nome: string,
             Luoghi: set(Luogo),
             Costo: integer);

add class Teatro inherits Monumento
  type tuple(SerateSpettacoli: list(date))

add class SpettacoloTeatrale
  type tuple(Titolo: string,
             Luogo: Teatro,
             Protagonista: Persona,
             Repliche: set(date));

add class Concerto inherits SpettacoloTeatrale
  type tuple(Protagonista: Direttore,
             Orchestra:set(Musicisti));

add class Persona
  type tuple(Nome: string,
             CodiceFiscale: string,
             Nazione: string);
```

```
add class Direttore inherits Persona
  type tuple(AffiliazioneStabile: Teatro);

add class Musicista inherits Persona
  type tuple(Strumenti: set(string));
```

1. Definire i metodi di inizializzazione delle tre classi Luogo, Monumento e Teatro, riutilizzando i metodi nello scendere lungo la gerarchia.
2. Quale proprietà dello schema è ridefinita in modo co-variante?
3. Definire la segnatura del metodo di inizializzazione di uno spettacolo e poi raffinare la segnatura in modo co-variante nei parametri di ingresso qualora lo spettacolo sia un concerto.
4. Indicare un esempio di invocazione del metodo definito in precedenza in cui non sia possibile verificarne la correttezza a tempo di compilazione.
5. Descrivere graficamente lo schema precedente nel modo illustrato in Figura 20.2.

20.4 Descrivere lo schema di base di dati a oggetti dell'Esercizio 20.3 usando la sintassi di SQL-3.

20.5 Considerando lo schema SQL-3 introdotto nell'Esercizio precedente, esprimere le seguenti interrogazioni in SQL-3.

1. Estrarre i nomi delle città la cui nazione è il "Liechtenstein".
2. Estrarre il nome dei musicisti che suonano in concerti diretti da Karajan.
3. Estrarre il nome dei monumenti di Londra costruiti nel diciassettesimo secolo e chiusi al lunedì.
4. Estrarre i nomi dei direttori che hanno diretto concerti in teatri diversi da quelli da cui dipendono.

20.6 Facendo riferimento allo schema di base di dati a oggetti dell'Esercizio 20.3, indicare una scelta di indici complessi per gestire in modo efficiente i cammini più usati dalle interrogazioni dell'Esercizio 20.5.

20.7 Costruire una rappresentazione a *2d-tree* e a *quad-tree* della sequenza di punti bidimensionali: $A(5,4)$, $B(3,3)$, $C(6,2)$, $D(2,2)$, $E(4,6)$, $F(1,1)$, $G(7,5)$. Quanti nodi intermedi vi sono, nelle due rappresentazioni, tra A e F oppure tra A e G?