Capitolo 7

Diagrammi UML delle classi

In questo capitolo volgiamo la nostra attenzione verso uno dei più importanti documenti prodotti nella specifica del software: il diagramma UML delle classi. Nato a metà degli anni '90 come unificazione di precedenti linguaggi per l'analisi orientata agli oggetti, UML [23, 18] è probabilmente il più importante linguaggio per la modellazione attualmente usato nell'ingegneria del software. Il diagramma delle classi rappresenta il principale documento UML per la descrizione degli aspetti *statici* dei programmi, ovvero quelli che prescindono dall'esecuzione. Dal punto di vista cronologico, si tratta di un diagramma prodotto durante l'*analisi*, ovvero nelle fasi alte del ciclo di vita. In altri capitoli di questa parte considereremo altri diagrammi UML, che appartengono alla categoria dei diagrammi *comportamentali*.

7.1 Sintassi dei diagrammi UML delle classi

Il diagramma UML delle classi offre la possibilità di modellare molti aspetti che sono potenzialmente di interesse per varie fasi dello sviluppo del software. In questo capitolo ci concentriamo sull'uso di questo tipo di diagramma dal punto di vista puramente *concettuale*. Pur tenendo presente che alcuni autori, ad es. [18], propongono l'uso del diagramma delle classi anche in altre fasi, ad esempio nell'implementazione, noi ignoreremo alcune delle potenzialità del linguaggio.

In particolare, prenderemo in considerazione gli aspetti dei diagrammi delle classi elencati nel seguito.

Classi: Le classi UML rappresentano insiemi di oggetti con aspetti comuni. Una classe viene rappresentata graficamente mediante un rettangolo diviso in tre parti. La parte superiore (l'unica obbligatoria) contiene il *nome* della classe, che deve essere unico nel diagramma. Le altre parti contengono, rispettivamente gli *attributi* e le *operazioni* della classe.

- **Attributi:** Gli attributi rappresentano proprietà *locali* degli oggetti. Ignoreremo i qualificatori degli attributi, ad es. quelli che li distinguono fra pubblici, protetti e privati. I nomi degli attributi sono unici all'interno di una classe, ma due classi differenti possono avere attributi con lo stesso nome.
- **Tipi degli attributi:** Ogni attributo ha un *tipo* associato ad esso. Un tipo è una collezione di *valori*, a cui non viene associato un ciclo di vita (come si fa con gli oggetti). Come tipi vengono solitamente utilizzate entità che hanno una diretta implementazione nei linguaggi di programmazione, ad es. *intero*, *reale*, *stringa*, ...
- **Molteplicità degli attributi:** Una molteplicità opzionale $\{i..j\}$ per un certo attributo di tipo T specifica che ogni oggetto della classe è associato ad almeno i ed al più j istanze di T. Quando la molteplicità non è esplicitamente riportata l'attributo è *monovalore*, ovvero viene assunto implicitamente $\{1..1\}$. Quando non c'è limite inferiore, si usa 0 al posto di i. Quando non c'è limite superiore, si usa $*$ al posto di j.
- **Operazioni:** Un'operazione è una funzione dagli oggetti della classe a cui l'operazione è associata a un risultato (opzionale) che può appartenere a una classe o a un tipo. L'operazione può avere altri argomenti oltre all'oggetto su cui viene invocata; tali argomenti possono appartenere a classi o a tipi.

  Le *precondizioni* e le *postcondizioni* delle operazioni, che ne caratterizzano in maniera precisa il significato, verranno prese in considerazione nei capitoli successivi.

La figura 7.1 mostra una classe con tre attributi, di cui due monovalore ed uno (*NumTel*) con almeno un valore e senza limite superiore al numero di valori. La classe ha anche due operazioni, entrambe con risultato, una senza argomenti ed una (*NumeroEsami()*) con un argomento di tipo *intero*, che rappresenta l'anno di corso.

Associazioni: Un'associazione è una relazione fra una o più classi ed ha un nome unico nel diagramma. Le associazioni *binarie* sono rappresentate come in figura 7.2, e possono presentare *vincoli di molteplicità* come gli attributi delle classi. Nella figura viene specificato che ogni studente può avere sostenuto l'esame per un numero di corsi compresi fra 0 e 3, mentre l'esame di un corso può essere stato sostenuto da un numero arbitrario di studenti.

Le associazioni possono avere un'arità arbitraria, purché maggiore di uno. Nel caso di associazioni con arità maggiore di due (cfr. in figura 7.3 un'associazione ternaria) i vincoli di molteplicità tipicamente non vengono specificati.

Per semplicità, nell'ambito di questo capitolo non prenderemo in considerazione le cosiddette *classi associazione* e dedicheremo agli *attributi di associazione* solo alcune considerazioni al termine del paragrafo 7.2.

Generalizzazioni: Una generalizzazione fra una *superclasse* e una *sottoclasse* esprime che ogni istanza di quest'ultima è anche un'istanza della prima (relazione *ISA*). Di conseguenza, le istanze della sottoclasse *ereditano* le proprietà della superclasse, e ne possono avere di specifiche. Ad esempio, in figura 7.4 la sottoclasse *LibroStorico* eredita dalla superclasse *Libro* l'attributo *Titolo* e ha un attributo specifico dal nome *Epoca*.

Gerarchie: In alcuni casi si desidera esprimere esplicitamente che le classi facenti parte di una gerarchia soddisfano ulteriori vincoli. Ad esempio, nella figura 7.5 la dicitura {*disjoint*} rappresenta il fatto che nessuna istanza della classe *Ufficiale* appartiene alla classe *Sottufficiale* o *MilitareDiTruppa* (e lo stesso dicasi per le altre combinazioni). La dicitura {*complete*} rappresenta il fatto che ogni istanza della classe *Militare* appartiene necessariamente ad una delle tre sottoclassi.

Specializzazioni: Nell'ambito delle generalizzazioni, le sottoclassi possono *specializzare* proprietà ereditate dalla superclasse.

Specializzazione di attributi: Se una classe C_1 ha un attributo A di tipo T_1, e se C_2 è una sottoclasse di C_1, specializzare A in C_2 significa definire A anche in C_2 ed assegnargli un tipo T_2 i cui valori sono un sottoinsieme dei valori di T, e/o restringerne i vincoli di molteplicità. Ad esempio, in figura 7.6 affermiamo che l'attributo *Età* viene specializzato (con un tipo più ristretto) nella sottoclasse *Anziano* (mantenendo il vincolo di molteplicità {1..1}).

Specializzazione di associazioni: La cosiddetta *specializzazione di associazione* stabilisce che tutte le tuple (o *link*) di una certa associazione facciano necessariamente parte di un'altra. Le associazioni coinvolte devono avere la stessa arità. Spesso la specializzazione fra associazioni prevede una o più generalizzazioni fra classi. Ad esempio, in figura 7.7 affermiamo che l'associazione *dedicatoA* specializza *presente*, ovvero che ogni coppia di oggetti (di tipo *Personaggio* e *Ritratto*) legati dall'associazione *dedicatoA* deve essere legata anche dall'associazione *presente*.

Vincoli: I vincoli rappresentano ulteriori informazioni che non possono essere rappresentate in maniera diretta mediante la sintassi grafica del diagramma UML delle classi. Tipicamente, per esprimere queste informazioni viene utilizzato il linguaggio OCL (cfr. [23]) acronimo di *object constraint language*). Per non introdurre un nuovo linguaggio, in questo capitolo faremo uso della logica del prim'ordine, di cui OCL può essere considerato un "dialetto". Per il momento, ci limitiamo ad elencare alcuni dei vincoli che potremmo essere interessati a rappresentare. L'elenco non è esaustivo, e nel resto del capitolo verranno forniti ulteriori esempi.

Contenimento fra tipi: ad esempio, che *interoPos* è strettamente contenuto in *intero* (cfr. figura 7.1).

Dipendenze funzionali: ad esempio, che uno studente può effettuare l'esame per un certo corso al più una volta, a prescindere dal professore coinvolto (cfr. figura 7.3).

Identificatori di classe: ad esempio, che la matricola identifica univocamente uno studente (cfr. figura 7.1).

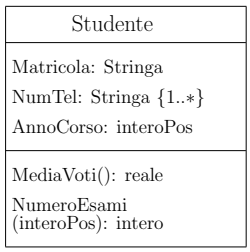

Figura 7.1 Un diagramma UML delle classi

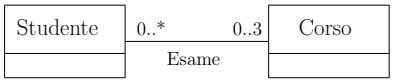

Figura 7.2 Un'associazione binaria

7.2 Semantica dei diagrammi UML delle classi

In questo paragrafo specificheremo la semantica dei diagrammi UML delle classi, fornendo la traduzione in logica del prim'ordine di ognuno degli elementi sintattici specificati nel paragrafo precedente.

Classi: Una classe UML di nome C viene rappresentata mediante un simbolo di predicato unario $C/1$.

Attributi: Un attributo a viene rappresentato mediante un simbolo di predicato binario $a/2$.

Tipi degli attributi: Un tipo T viene rappresentato mediante un simbolo di predicato unario $T/1$.

La formula:

$$\forall XY\ C(X) \wedge a(X,Y) \rightarrow T(Y),$$

rappresenta il fatto che la classe C ha un attributo di nome a e tipo T.

Molteplicità degli attributi: I vincoli di molteplicità possono essere formalizzati mediante una formula del prim'ordine che caratterizzi i limiti inferiore e superiore della cardinalità dell'insieme di istanze del tipo T coinvolte mediante l'attributo a.

Per quanto riguarda la specifica del limite inferiore i, ci serviamo di formule esemplificate dalla seguente, valida per $i = 2$:

$$\forall X\ (C(X) \rightarrow \exists YZ\ a(X,Y) \wedge a(X,Z) \wedge Y \neq Z). \tag{7.1}$$

Il numero di variabili quantificate esistenzialmente è pari a i, mentre il numero di diseguaglianze ($\neq$) cresce come $\Theta(i^2)$.

Per quanto riguarda la specifica del limite superiore j, ci serviamo di formule esemplificate dalla seguente, valida per $j = 2$:

$$\forall XYZW\ ((C(X) \wedge a(X,Y) \wedge a(X,Z) \wedge a(X,W)) \rightarrow (Y = Z \vee Y = W \vee Z = W)). \tag{7.2}$$

Il numero di variabili quantificate universalmente è pari a $j+2$, mentre il numero di uguaglianze ($=$) cresce come $\Theta(j^2)$.

Nel caso di limite inferiore pari a zero o limite superiore pari a infinito ('*'), le formule (7.1) o (7.2), rispettivamente, non vanno indicate.

In alcuni testi, per esprimere in maniera compatta una molteplicità $\{i..j\}$ si fa uso della notazione seguente:

$$\forall X\ C(X) \rightarrow (i \leq \sharp\{Y \mid a(X,Y)\} \leq j),$$

dove $\sharp$ denota la cardinalità di un insieme.

Operazioni: Un'operazione va rappresentata fornendo formule del prim'ordine che ne caratterizzino la *segnatura*, ovvero il nome, il numero e il tipo o classe degli argomenti, e il tipo o classe dell'eventuale risultato.

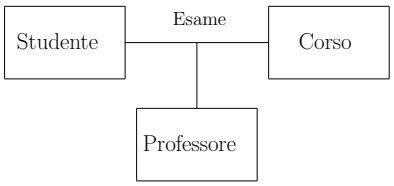

Figura 7.3 Un'associazione ternaria

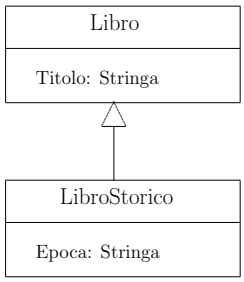

Figura 7.4 Una generalizzazione

Un'operazione *o()* della classe C con n argomenti il cui tipo o classe è rappresentato mediante i predicati unari $P_1, \ldots, P_n$, rispettivamente, e il cui tipo o classe del risultato è rappresentato, se presente, mediante il predicato unario R, viene rappresentata mediante un simbolo di predicato o. L'arità di o è pari a $n+2$ se l'operazione fornisce risultato, è pari a $n+1$ altrimenti. Il primo argomento di o sta per l'oggetto d'invocazione, i successivi per gli argomenti e l'ultimo per il risultato, se presente. Si noti che se il nome dell'operazione non è unico nell'ambito del diagramma, allora vanno usati differenti simboli di predicato per ognuna di esse. L'operazione $o()$ ha associate le seguenti formule:

$$\forall X A_1 \ldots A_n B \qquad o(X, A_1 \ldots A_n, B) \rightarrow (C(X) \wedge R(B) \wedge \bigwedge_{i=1}^{n} P_i(A_i)), \tag{7.3}$$

$$\forall X A_1 \ldots A_n B_1 B_2 \qquad o(X, A_1 \ldots A_n, B_1) \wedge o(X, A_1 \ldots A_n, B_2) \rightarrow B_1 = B_2. \tag{7.4}$$

Le formule (7.3-7.4) caratterizzano il caso in cui $o()$ fornisca un risultato. La (7.3) impone agli argomenti di o il tipo o classe di pertinenza, mentre la (7.4) specfica che l'operazione è una *funzione*, ovvero restituisce un solo valore. Le formule relative al caso in cui $o()$ non fornisca risultato sono simili, e non contengono occorrenze del predicato R.

La figura 7.8 mostra le formule del prim'ordine relative alla classe UML della figura 7.1.

Associazioni: Un'associazione UML A di arità n viene rappresentata mediante un simbolo di predicato n-ario A/n. Se le classi che sono interessate dall'associazione A sono rappresentate, rispettivamente, dai predicati unari $C_1, \ldots, C_n$, la formula

$$\forall X_1 \ldots X_n \; A(X_1, \ldots, X_n) \rightarrow (C_1(X_1) \wedge \cdots C_n(X_n)),$$

rappresenta il fatto che l'associazione insiste su tali classi. Ad esempio, per l'associazione della figura 7.3 usiamo la seguente formula:

$$\forall XYZ \qquad Esame(X, Y, Z) \rightarrow Studente(X) \wedge Corso(Y) \wedge Professore(Z).$$

I vincoli di molteplicità delle associazioni binarie vengono rappresentati come i vincoli di molteplicità degli attributi delle classi, tenendo conto che va presa in considerazione una classe per volta. Ad esempio, l'unico vincolo della figura 7.2 che va rappresentato è il seguente:

$$\forall X \; Studente(X) \rightarrow (0 \leq \sharp\{Y \mid Esame(X, Y)\} \leq 3).$$

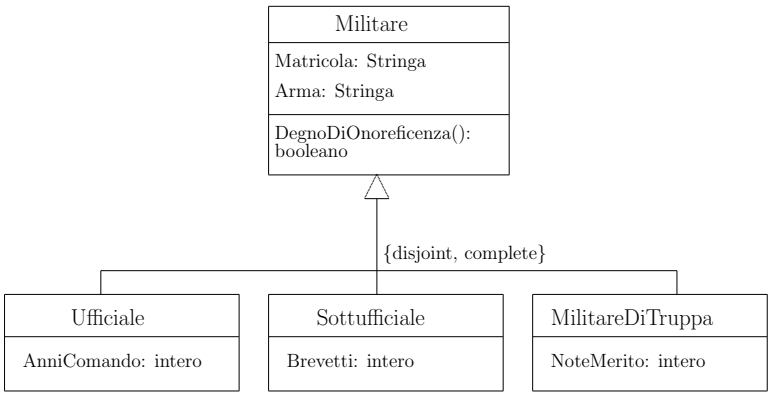

Figura 7.5 Una generalizzazione disgiunta e completa

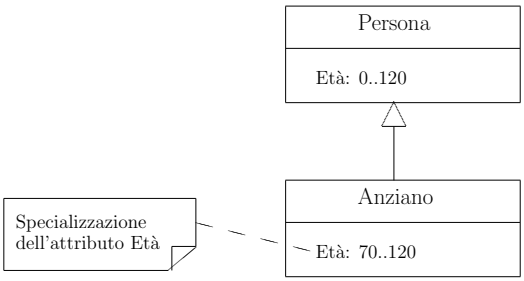

Figura 7.6 Specializzazione di attributo

Esercizio 7.1 [Soluzione a pag. 112] Rappresentare in maniera completa e non abbreviata tutte le formule relative al diagramma della figura 7.2. ◦

Generalizzazioni: Una generalizzazione fra una superclasse D e una sottoclasse C viene rappresentata mediante la formula

$$\forall X \; C(X) \rightarrow D(X). \tag{7.5}$$

Gerarchie: In una gerarchia che coinvolge una superclasse D e n sottoclassi $C_1, \ldots, C_n$, il vincolo $\{complete\}$ viene rappresentato mediante la formula:

$$\forall X \; D(X) \rightarrow \bigvee_{i=1}^{n} C_i(X), \tag{7.6}$$

mentre il vincolo $\{disjoint\}$ mediante la seguente formula, costituita da $\frac{n \times (n-1)}{2}$ congiunzioni:

$$\bigwedge_{i=1}^{n} \bigwedge_{j<i} \forall X \; C_i(X) \rightarrow \neg C_j(X). \tag{7.7}$$

Esercizio 7.2 Rappresentare in maniera completa tutte le formule relative ai diagrammi delle figure 7.4 e 7.5. ◦

Specializzazioni:

Specializzazione di attributi: La specializzazione di attributi non ha bisogno di essere rappresentata mediante formule ulteriori oltre a quelle viste in precedenza (dato che l'attributo specializzato è stato modellato esplicitamente nella sottoclasse).

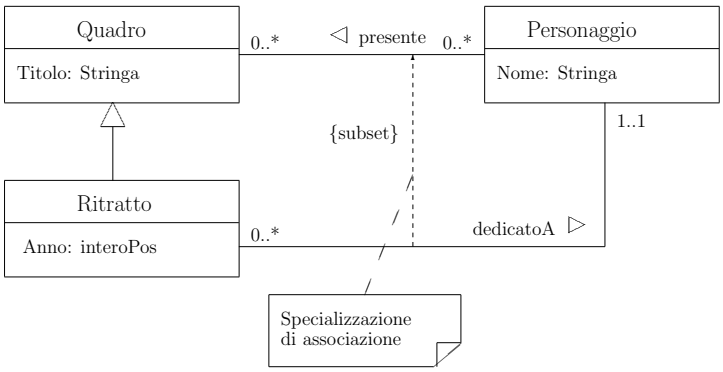

Figura 7.7 Specializzazione fra associazioni

Specializzazione di associazioni: La specializzazione dell'associazione A mediante l'associazione B, entrambe di arità $n \geq 2$, viene rappresentata mediante la formula:

$$\forall X_1, \ldots, X_n \; B(X_1, \ldots, X_n) \rightarrow A(X_1, \ldots, X_n).$$

Con riferimento alla figura 7.7, la specializzazione viene rappresentata mediante la formula:

$$\forall XY \; \mathit{presente}(X, Y) \rightarrow \mathit{dedicatoA}(X, Y).$$

Esercizio 7.3 Rappresentare in maniera completa tutte le formule relative al diagramma delle figure 7.6 e 7.7. ◦

Vincoli: Facendo riferimento agli esempi menzionati nel paragrafo 7.1, riportiamo le formule del prim'ordine corrispondenti.

Contenimento fra tipi: *interoPos* è strettamente contenuto in *intero*:

$$\begin{aligned} &\forall X \; \mathit{interoPos}(X) \rightarrow \mathit{intero}(X) \; \wedge \\ &\quad \exists X \; \neg \mathit{interoPos}(X) \wedge \mathit{intero}(X). \end{aligned}$$

Si noti che una formula simile deve essere usata anche per il diagramma della figura 7.6.

Dipendenze funzionali: uno studente può effettuare l'esame per un certo corso al più una volta, a prescindere dal professore coinvolto (cfr. figura 7.3):

$$\forall XYZW \qquad \mathit{Esame}(X, Y, Z) \; \wedge \; \mathit{Esame}(X, Y, W) \rightarrow Z = W.$$

Identificatori di classe: la matricola identifica univocamente uno studente (cfr. figura 7.1):

$$\forall XYZ \quad \mathit{Studente}(X) \; \wedge \; \mathit{Studente}(Y) \; \wedge \; \mathit{Matricola}(X, Z) \; \wedge \; \mathit{Matricola}(Y, Z) \rightarrow X = Y.$$

Esercizio 7.4 [Soluzione a pag. 112] Rappresentare in maniera completa tutte le formule relative al diagramma della figura 7.9. ◦

Prima di concludere questo paragrafo vogliamo prendere in considerazione due ulteriori aspetti. Il primo riguarda una possibile estensione del linguaggio dei diagrammi delle classi. Nel paragrafo 7.1, infatti, non abbiamo preso in considerazione i cosiddetti *attributi di associazione*, ovvero valori che possono essere associati alle tuple di un'associazione. La figura 7.10 mostra un diagramma in cui interessa, oltre all'associazione *Sindaco* fra le classi *Persona* e *Città*, anche l'anno di elezione; quest'ultimo dipende sia dalla persona sia dalla città, e quindi non può essere attributo di nessuna delle due classi, ma è appunto un attributo di associazione. Si noti che l'associazione nel diagramma è binaria, e quindi esiste al più una tupla che lega una data persona e una data città, a prescindere dall'anno di elezione.

$$
\begin{array}{rl}
 & \text{// attributo } \textit{Matricola} \\
\forall XY & Studente(X) \land Matricola(X,Y) \rightarrow Stringa(Y), \\
\forall X & Studente(X) \rightarrow \exists Y\ Matricola(X,Y), \\
\forall XYZ & Studente(X) \land Matricola(X,Y) \land Matricola(X,Z) \rightarrow Y = Z, \\
 & \text{// attributo } \textit{NumTel} \\
\forall XY & Studente(X) \land NumTel(X,Y) \rightarrow Stringa(Y), \\
\forall X & Studente(X) \rightarrow \exists Y\ NumTel(X,Y), \\
 & \text{// attributo } \textit{AnnoCorso} \\
\forall XY & Studente(X) \land AnnoCorso(X,Y) \rightarrow InteroPos(Y), \\
\forall X & Studente(X) \rightarrow \exists Y\ AnnoCorso(X,Y), \\
\forall XYZ & Studente(X) \land AnnoCorso(X,Y) \land AnnoCorso(X,Z) \rightarrow Y = Z, \\
 & \text{// operazione } \textit{MediaVoti()} \\
\forall XY & MediaVoti(X,Y) \rightarrow Studente(X) \land Reale(Y), \\
\forall XYZ & MediaVoti(X,Y) \land MediaVoti(X,Z) \rightarrow Y = Z, \\
 & \text{// operazione } \textit{NumeroEsami()} \\
\forall XYZ & NumeroEsami(X,Y,Z) \rightarrow Studente(X) \land InteroPos(Y) \land Reale(Z), \\
\forall XYWZ & NumeroEsami(X,Y,W) \land NumeroEsami(X,Y,Z) \rightarrow W = Z.
\end{array}
$$

Figura 7.8 Formule del prim'ordine relative alla classe di figura 7.1

La formalizzazione degli atributi di associazione può essere fatta in vari modi. Una maniera semplice consiste nell'utilizzare, per la rappresentazione di un'associazione con attributi, un simbolo di predicato di arità pari al numero di classi coinvolte più il numero di attributi di associazione. Ad esempio, potremmo usare un predicato di arità tre per l'associazione *Sindaco* di figura 7.10. Poiché gli attributi di associazione sono determinati in maniera funzionale dagli oggetti (ad esempio, non è possibile legare una persona p e una città c mediante due tuple con anni di elezione a_1 e a_2 differenti), sono necessarie opportune formule (simili alla 7.4) che esprimano tale vincolo.

Esercizio 7.5 Progettare un metodo generale per rappresentare associazioni con attributi mediante formule della logica del prim'ordine. Applicare tale metodo al diagramma della figura 7.10. ◦

L'altro aspetto che vogliamo discutere riguarda l'unicità delle classi di appartenenza degli oggetti. Per comodità articoliamo questo aspetto in due casi distinti: diagrammi delle classi senza e con gerarchie. Nei primi si fa spesso l'ipotesi che ogni oggetto appartenga ad una sola classe. Questa ipotesi può essere formalizzata, in un diagramma senza tipi in cui occorrono le classi $C_1, \ldots, C_n$, mediante le seguenti formule:

$$\forall X \bigvee_{i=1}^{n} C_i(X), \tag{7.8}$$

$$\bigwedge_{i=1}^{n} \bigwedge_{j<i} \forall X\ C_i(X) \rightarrow \neg C_j(X). \tag{7.9}$$

Si noti che le formule (7.8) e (7.9) sono simili alle formule (7.6) e (7.7), che rappresentano, rispettivamente, i vincoli *complete* e *disjoint* di una gerarchia. Nel caso in cui occorrano tipi dovremmo modificare le formule di cui sopra; ad esempio, per formalizzare il fatto che i tipi sono disgiunti dalle classi, dovremmo modificare la formula (7.9) aggiungendo ulteriori vincoli.

Quando il diagramma contiene generalizzazioni ovviamente non è più vero che ogni oggetto appartiene ad una sola classe: ad esempio, facendo riferimento alla figura 7.4, ogni oggetto della classe *LibroStorico* appartiene anche alla classe *Libro*. In tal caso, si fanno altre due ipotesi:

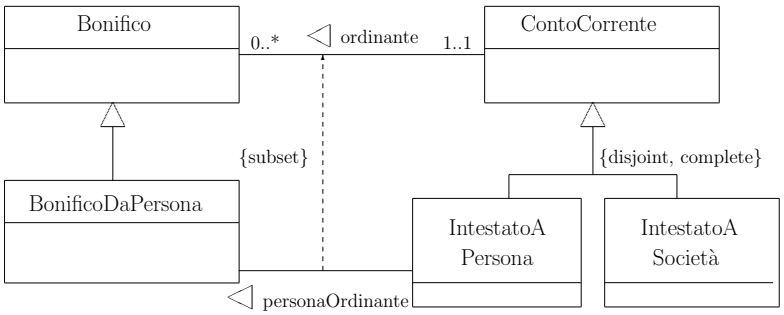

Figura 7.9 Specializzazione fra associazioni e vincoli di molteplicità

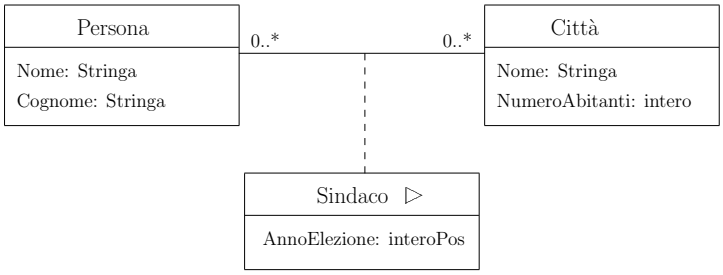

Figura 7.10 Attributi di associazione

- che le classi non appartenenti ad una stessa gerarchia siano disgiunte,
- che ogni oggetto di una gerarchia appartenga ad un'unica classe più specifica.

In altre parole, le classi sono implicitamente disgiunte, e possono avere oggetti in comune solo se hanno una sottoclasse in comune. Ad esempio, nel diagramma della figura 7.11 le classi *Mammifero* e *Acquatico* possono avere oggetti in comune, ma solo se tali oggetti appartengono alla sottoclasse comune *Cetaceo*. Per evitare che possano esistere oggetti che appartengono a due classi prive di sottoclassi comuni è necessario introdurre opportune formule. Nel caso del diagramma della figura 7.11 le formule sono le seguenti:

$$\forall X\ Mammifero(X) \wedge Acquatico(X) \rightarrow Cetaceo(X),$$
$$\forall X\ Rettile(X) \rightarrow \neg Acquatico(X),$$
$$\forall X\ Rettile(X) \rightarrow \neg Mammifero(X),$$

oltre alle formule, come

$$\forall X\ Mammifero(X) \rightarrow Animale(X),$$

che rappresentano le relazioni ISA della gerarchia.

7.3 Proprietà dei diagrammi UML delle classi

Nel paragrafo precedente abbiamo presentato la semantica dei diagrammi delle classi UML, fornendo la traduzione in logica del prim'ordine di ognuno degli elementi sintattici specificati nel paragrafo 7.1. In questo paragrafo vogliamo fornire un elenco, anche se parziale, del tipo di proprietà dei diagrammi delle classi a cui potremmo essere interessati. Faremo riferimento ad un generico diagramma UML delle classi D a cui è associato un insieme di vincoli V; l'insieme di formule del prim'ordine che fornisce la semantica di D e V secondo la metodologia esposta nel paragrafo 7.2 sarà denotato con Φ.

Conseguenze implicite: Una proprietà, rappresentata da una formula del prim'ordine γ, è una *conseguenza implicita* di D e V se γ è una conseguenza logica di Φ, ovvero se $\Phi \models \gamma$ (cfr. paragrafo 4.3).

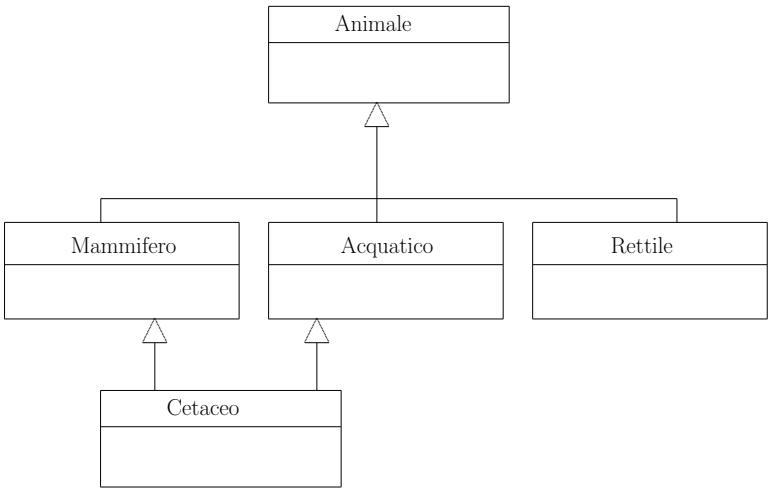

Figura 7.11 Una gerarchia di classi

Consistenza di una singola classe: Una classe C di D è *consistente* (o *soddisfacibile*) se D ammette un'istanziazione in cui C ha un numero maggiore di zero di oggetti, ovvero quando non è una conseguenza logica di Φ che C sia vuota, ovvero quando vale:

$$\Phi \not\models \forall X\ \neg C(X),$$

ovvero quando esiste un modello M di Φ in cui $M(C) \neq \emptyset$.

Chiaramente un diagramma con classi inconsistenti è scarsamente leggibile. L'inconsistenza di una classe è tipicamente dovuta ad un errore nella fase di analisi o in quella (precedente) di raccolta dei requisiti, che spesso consiste nell'aver posto troppi vincoli.

Sussunzione fra classi: Una classe C_1 *sussume* una classe C_2 se è una conseguenza implicita di D e V che sussista la relazione ISA fra C_2 e C_1, ovvero quando vale

$$\Phi \models \forall X\ C_2(X) \rightarrow C_1(X).$$

La sussunzione fra classi potrebbe denotare l'omissione di un'esplicita generalizzazione nel diagramma, oppure un errore.

Equivalenza fra classi: Quando esistono due classi C_1 e C_2 tali che C_1 sussume C_2 e viceversa, allora C_1 e C_2 si dicono *equivalenti*. In figura 7.12 le classi B, C e D sono equivalenti; tale conseguenza logica può essere dimostrata utilizzando formule del tipo (7.5).

Poiché due o più classi equivalenti hanno le stesse istanze, è auspicabile che il diagramma ne contenga una sola, per facilitarne la comprensione e ridurne la complessità.

Ristrutturazione di parti di diagramma: La ristrutturazione di un diagramma UML delle classi consiste in alcune modifiche che lo rendono migliore sotto diversi aspetti di qualità, come maggiore coesione, minore accoppiamento, migliore modularizzazione. Ovviamente il diagramma delle classi ristrutturato deve risultare equivalente a quello iniziale, ovvero deve permettere di modellare esattamente lo stesso insieme di livelli estensionali. Ad esempio, la ristrutturazione del diagramma delle classi di figura 7.13 consiste in una più adeguata associazione degli attributi alle classi, che comporta una maggiore coesione e quindi una modularizzazione migliore.

Esercizio 7.6 Dimostrare che le classi B, C e D della figura 7.12 sono equivalenti e riprogettare il diagramma di conseguenza. ◦

Esercizio 7.7 [Soluzione a pag. 113] Dimostrare che i due diagrammi di figura 7.13 sono equivalenti. ◦

Riportiamo ora alcuni esempi di verifica delle proprietà menzionate.

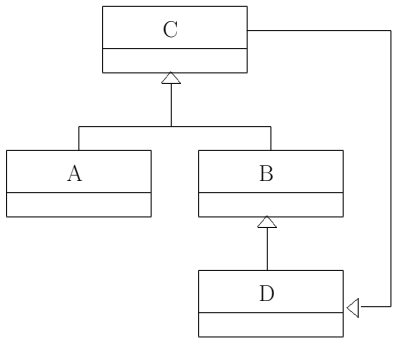

Figura 7.12 Una gerarchia ciclica

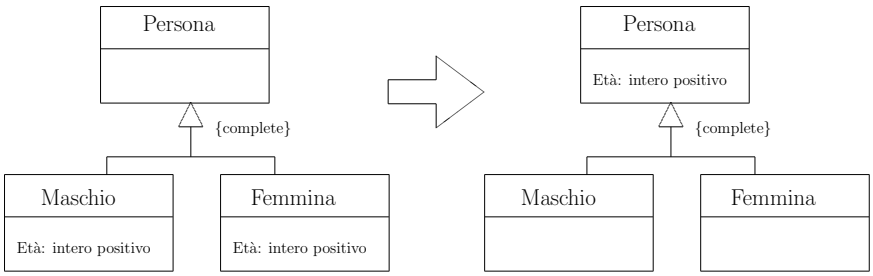

Figura 7.13 Ristrutturazione di un diagramma UML delle classi

Esempio 7.3.1 Facendo riferimento al diagramma della figura 7.4, la classe *LibroStorico* eredita l'attributo *Titolo*. Le formule del prim'ordine, denotate con Φ, relative al diagramma sono le seguenti:

$$
\begin{array}{rl}
 & \text{// classe } \mathit{Libro} \\
\forall XY & \mathit{Libro}(X) \wedge \mathit{Titolo}(X,Y) \rightarrow \mathit{Stringa}(Y), \\
\forall X & \mathit{Libro}(X) \rightarrow \exists Y\ \mathit{Titolo}(X,Y), \\
\forall XYZ & \mathit{Libro}(X) \wedge \mathit{Titolo}(X,Y) \wedge \mathit{Titolo}(X,Z) \rightarrow Y = Z, \\
 & \text{// classe } \mathit{LibroStorico} \\
\forall XY & \mathit{LibroStorico}(X) \wedge \mathit{Epoca}(X,Y) \rightarrow \mathit{Stringa}(Y), \\
\forall X & \mathit{LibroStorico}(X) \rightarrow \exists Y\ \mathit{Epoca}(X,Y), \\
\forall XYZ & \mathit{LibroStorico}(X) \wedge \mathit{Epoca}(X,Y) \wedge \mathit{Epoca}(X,Z) \rightarrow Y = Z, \\
 & \text{// } \mathit{LibroStorico} \text{ ISA } \mathit{Libro} \\
\forall X & \mathit{LibroStorico}(X) \rightarrow \mathit{Libro}(X), \\
 & \text{// Le classi sono disgiunte dai tipi} \\
\forall X & \mathit{Libro}(X) \rightarrow \neg \mathit{Stringa}(X).
\end{array}
$$

Introduciamo alcune formule ulteriori:

$$
\begin{array}{rrl}
 & & \text{// } \mathit{LibroStorico} \text{ eredita l'attributo } \mathit{Titolo} \\
\gamma_1: & \forall XY & \mathit{LibroStorico}(X) \wedge \mathit{Titolo}(X,Y) \rightarrow \mathit{Stringa}(Y), \\
 & & \text{// L'attributo } \mathit{Titolo} \text{ è monovalore in } \mathit{LibroStorico} \\
\gamma_2: & \forall X & \mathit{LibroStorico}(X) \rightarrow \exists Y\ \mathit{Titolo}(X,Y), \\
\gamma_3: & \forall XYZ & \mathit{LibroStorico}(X) \wedge \mathit{Titolo}(X,Y) \wedge \mathit{Titolo}(X,Z) \rightarrow Y = Z, \\
 & & \text{// } \mathit{Libro} \text{ eredita l'attributo } \mathit{Epoca} \\
\gamma_4: & \forall XY & \mathit{Libro}(X) \wedge \mathit{Epoca}(X,Y) \rightarrow \mathit{Stringa}(Y).
\end{array}
$$

Valgono le seguenti proprietà:

$$\Phi \models \gamma_1 \wedge \gamma_2 \wedge \gamma_3, \tag{7.10}$$

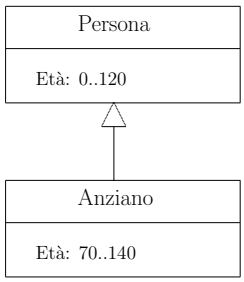

Figura 7.14 Un diagramma UML delle classi inconsistente

$$\Phi \not\models \gamma_4, \tag{7.11}$$

ovvero *LibroStorico* eredita (cfr. formula (7.10)) l'attributo *Titolo* con la stessa molteplicità, mentre *Libro* non eredita (cfr. formula (7.11)) l'attributo *Epoca*. ◦

Esempio 7.3.2 Facendo riferimento al diagramma della figura 7.9, siano Φ le formule del prim'ordine ad esso relative (cfr. esercizio 7.4). Introduciamo alcune formule ulteriori:

$$\begin{array}{lll} & & \text{// } \textit{BonificoDaPersona} \text{ eredita l'associazione } \textit{Ordinante} \\ \gamma_1: & \forall XY & BonificoDaPersona(X) \wedge Ordinante(X,Y) \rightarrow ContoCorrente(Y), \\ & & \text{// } \textit{BonificoDaPersona} \text{ eredita anche i vincoli di molteplicità sull'associazione ereditata} \\ \gamma_2: & \forall X & BonificoDaPersona(X) \rightarrow \exists Y\ Ordinante(X,Y), \\ \gamma_3: & \forall XYZ & BonificoDaPersona(X) \wedge Ordinante(X,Y) \wedge Ordinante(X,Z) \rightarrow Y = Z. \end{array}$$

Vale la seguente proprietà:

$$\Phi \models \gamma_1 \wedge \gamma_2 \wedge \gamma_3, \tag{7.12}$$

ovvero la classe *BonificoDaPersona* eredita l'associazione *Ordinante*, con i suoi vincoli di molteplicità.

Per quanto riguarda i vincoli di molteplicità dell'associazione *PersonaOrdinante*, essa eredita da *Ordinante* solo il limite superiore, ma non quello inferiore. Considerando infatti le ulteriori formule:

$$\begin{array}{lll} \gamma_4: & \forall XYZ & PersonaOrdinante(X,Y) \wedge PersonaOrdinante(X,Z) \rightarrow Y = Z, \\ \gamma_5: & \forall X & BonificoDaPersona(X) \rightarrow \exists Y\ PersonaOrdinante(X,Y), \end{array}$$

valgono le seguenti proprietà:

$$\Phi \models \gamma_4, \tag{7.13}$$
$$\Phi \not\models \gamma_5, \tag{7.14}$$

ovvero il vincolo di molteplicità di *PersonaOrdinante* è implicitamente 0..1, e non 1..1 come si potrebbe pensare. ◦

Esercizio 7.8 [Soluzione a pag. 113] Facendo riferimento al diagramma della figura 7.9, esprimere tramite opportune formule che:

1. ogni oggetto di classe *BonificoDaPersona* è legato tramite l'associazione *Ordinante* solamente a istanze della classe *IntestatoAPersona*, e
2. ogni oggetto di classe *IntestatoAPersona* è legato tramite l'associazione *Ordinante* solamente a istanze della classe *BonificoDaPersona*.

Tali formule sono conseguenze logiche della formula Φ che traduce il diagramma? In caso negativo, cosa fareste? ◦

Esercizio 7.9 Dimostrare che la classe *Anziano* del diagramma della figura 7.14 è inconsistente. ◦

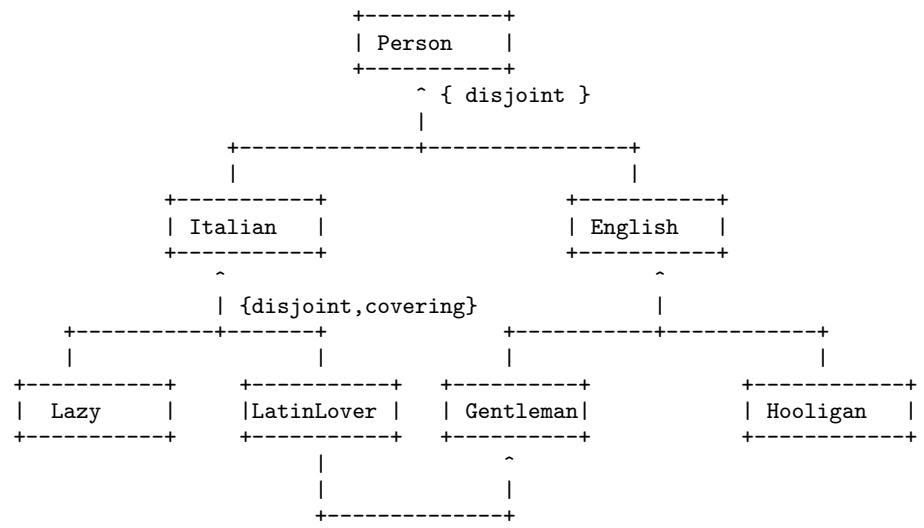

Figura 7.15 Una gerarchia di classi

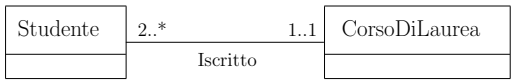

Figura 7.16 Vincoli di molteplicità che limitano il numero di studenti possibili

Esercizio 7.10 [Soluzione a pag. 114] Facendo riferimento al diagramma della figura 7.15 (esempio del Prof. E. Franconi), esistono classi inconsistenti? In caso affermativo, cosa occorre eliminare per ristabilire la consistenza? ◦

Esempio 7.3.3 Facendo riferimento al diagramma della figura 7.16, sia Φ l'insieme delle formule del prim'ordine ad esso relative. I vincoli di molteplicità limitano il numero di studenti possibili: possono esistere zero oppure un numero maggiore o uguale a due studenti, ma non uno. Considerando infatti le ulteriori formule:

$$\begin{aligned} \gamma_1: &\quad \exists X \quad Studente(X), \\ \gamma_2: &\quad \forall XY \quad Studente(X) \wedge Studente(Y) \rightarrow X = Y, \end{aligned}$$

vale la seguente proprietà:

$$\Phi \models \neg(\gamma_1 \wedge \gamma_2). \tag{7.15}$$

◦

Esempio 7.3.4 L'esempio 7.3.3 mostra come i vincoli di molteplicità possano limitare il numero di oggetti di una certa classe. In alcuni casi non esiste alcun numero finito n tale che una classe possa avere esattamente n oggetti. Facendo riferimento al diagramma della figura 7.17, la classe *Studente* non può avere alcun numero di oggetti, se non zero o infiniti. La classe *Studente* si dice *finitamente inconsistente* (o *finitamente insoddisfacibile*). ◦

Esercizio 7.11 Esistono classi finitamente inconsistenti nel diagramma della figura 7.18? ◦

7.4 Aspetti computazionali

In questo capitolo abbiamo visto che una specifica fornita mediante diagramma UML delle classi, corredata o meno da un insieme di vincoli, conduce a una formula della logica del prim'ordine. Abbiamo

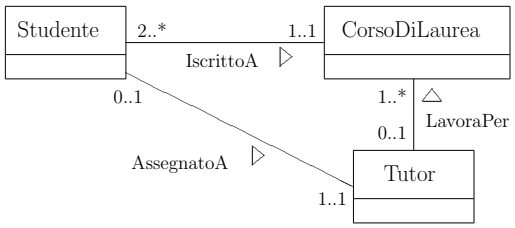

Figura 7.17 Diagramma delle classi non finitamente soddisfacibile

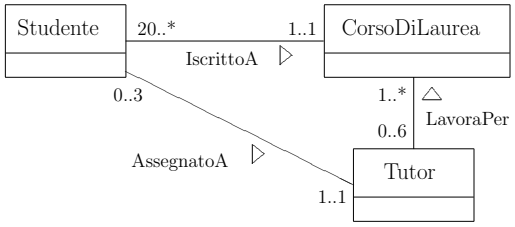

Figura 7.18 Un diagramma UML delle classi

inoltre appurato che molti ragionamenti volti alla modifica e al miglioramento del diagramma possono essere rappresentati mediante inferenze in logica del prim'ordine. In questo paragrafo ci concentriamo su alcuni aspetti computazionali delle nozioni viste, sfruttando le basi fornite nel capitolo 4.

7.4.1 Complessità computazionale

In presenza di vincoli arbitrari il ragionamento sui diagrammi delle UML classi non è decidibile, in quanto vale il risultato menzionato nel paragrafo 4.5.1.

In diagrammi senza vincoli, oltre a quelli espressi implicitamente in maniera grafica, il ragionamento è decidibile, e più precisamente è EXPTIME-completo (cfr. [3]).

7.4.2 Programmi disponibili

Come abbiamo avuto modo di vedere nel paragrafo 7.3, le proprietà più interessanti consistono nella verifica di implicazioni nella logica del prim'ordine. Di conseguenza, possiamo usare i programmi illustrati nel capitolo 4.

7.4.2.1 Verifica di proprietà tramite un dimostratore di teoremi

In questo paragrafo vedremo come usare OTTER per la verifica di conseguenze implicite.

Esempio 7.4.1 [continuazione dell'esempio 7.3.1] Possiamo facilmente dimostrare la validità di (7.10) e (7.11) mediante il seguente file.

```
set(auto).
formula_list(usable).

%% CLASSE LIBRO
all X Y (Libro(X) & Titolo(X,Y) -> Stringa(Y)).            % Tipo attributo
all X (Libro(X) -> (exists Y Titolo(X,Y))).                % Attributo {1..*}
all X Y Z (Libro(X) & Titolo(X,Y) & Titolo(X,Z) -> Y=Z).   % Attributo {0..1}

%% ISA
all X (LibroStorico(X) -> Libro(X)).

%% CLASSE LIBROSTORICO
all X Y (LibroStorico(X) & Epoca(X,Y) -> Stringa(Y)).        % Tipo attributo
all X (LibroStorico(X) -> (exists Y Epoca(X,Y))).            % Attributo {1..*}
all X Y Z (LibroStorico(X) & Epoca(X,Y) & Epoca(X,Z) -> Y=Z).% Attributo {0..1}
```

```
%% CLASSI DISGIUNTE DA TIPI
all X (Libro(X) -> -Stringa(X)).


%% CONSEGUENZE LOGICHE DA DIMOSTRARE:

%% LibroStorico eredita attributo "Titolo : Stringa {1..1}" (vero)
C1 <-> (all X Y (LibroStorico(X) & Titolo(X,Y) -> Stringa(Y))).
C2 <-> (all X (LibroStorico(X) -> (exists Y Titolo(X,Y)))).
C3 <-> (all X Y Z (LibroStorico(X) & Titolo(X,Y) & Titolo(X,Z) -> Y=Z)).
LibroStoricoEreditaTitolo <-> (C1 & C2 & C3).

%% Libro EREDITA ATTRIBUTO Epoca (falso)
LibroEreditaEpoca <-> (all X Y (Libro(X) & Epoca(X,Y) -> Stringa(Y))).

%% TESI
-(
   LibroStoricoEreditaTitolo
   %LibroEreditaEpoca  (falso, usare mace)
).

end_of_list.
```

Come sappiamo, per dimostrare la (7.10) è opportuno usare OTTER mentre per la (7.11) è opportuno cercare un controesempio mediante MACE. ◦

Esempio 7.4.2 [continuazione dell'esempio 7.3.2] Possiamo facilmente dimostrare la validità di (7.12), (7.13) e (7.14) mediante il seguente file.

```
set(auto).
formula_list(usable).

%% ASSOCIAZIONE ORDINANTE
all X Y (Ordinante(X,Y) -> Bonifico(X) & ContoCorrente(Y)).
all X (Bonifico(X) -> (exists Y Ordinante(X,Y))).            % {1..*}
all X Y Z (Ordinante(X,Y) & Ordinante(X,Z) -> Y=Z).          % {0..1}

%% ASSOCIAZIONE PERSONAORDINANTE
all X Y (PersonaOrdinante(X,Y) -> BonificoDaPersona(X) & IntestatoAPersona(Y)).

%% SUBSET FRA ASSOCIAZIONI
all X Y (PersonaOrdinante(X,Y) -> Ordinante(X,Y)).


%% ISA "BONIFICO"
all X (BonificoDaPersona(X) -> Bonifico(X)).

% ISA "CONTOCORRENTE"
all X (IntestatoAPersona(X) -> ContoCorrente(X)).
all X (IntestatoASocieta(X) -> ContoCorrente(X)).
all X (IntestatoASocieta(X) -> -IntestatoAPersona(X)).  % disjoint
all X (ContoCorrente(X) -> (IntestatoASocieta(X) | IntestatoAPersona(X))).  % complete


%% DISGIUNZIONE TRA CLASSI NON LEGATE DA GERARCHIE
all X (Bonifico(X) -> -ContoCorrente(X)).


%% CONSEGUENZE LOGICHE DA DIMOSTRARE:

%% BonificoDaPersona eredita l'associazione Ordinante...
C1 <-> (all X Y (BonificoDaPersona(X) & Ordinante(X,Y) -> ContoCorrente(Y))).
%% ...con i suoi vincoli di molteplicita' (1..1)
C2 <-> (all X (BonificoDaPersona(X) -> (exists Y Ordinante(X,Y)))).
C3 <-> (all X Y Z (BonificoDaPersona(X) & Ordinante(X,Y) & Ordinante(X,Z) ->  Y=Z)).
BonifDaPersonaEreditaOrdinante <-> (C1 & C2 & C3).
```

```
%% Vincoli di molteplicità associazione PersonaOrdinante (per il ruolo BonificoDaPersona)
BonifDaPersMoltMaxPersOrdin1 <-> (all X Y Z (PersonaOrdinante(X,Y) & PersonaOrdinante(X,Z) -> Y=Z)).
BonifDaPersMoltMinPersOrdin1 <-> (all X (BonificoDaPersona(X) -> (exists Y PersonaOrdinante(X,Y)))).


%% Ogni oggetto di classe BonificoDaPersona è legato tramite l'associazione
%% Ordinante solamente a istanze della classe IntestatoAPersona
BonifDaPersSoloACCIntAPers <->
(all X Y (BonificoDaPersona(X) & Ordinante(X,Y) -> IntestatoAPersona(Y))).

%% TESI
-(
   BonifDaPersonaEreditaOrdinante
   %BonifDaPersMoltMaxPersOrdin1
   %BonifDaPersMoltMinPersOrdin1  % (falso, usare mace)
   %BonifDaPersSoloACCIntAPers    % (falso, usare mace)
).

end_of_list.
```

L'ultima parte del file contiene formule utili per l'esercizio 7.8. ◦

Esempio 7.4.3 [continuazione dell'esempio 7.3.3] Possiamo facilmente dimostrare la validità di (7.15) mediante il seguente file.

```
set(auto).
formula_list(usable).

% ASSOCIAZIONE Iscritto (Studente 2..* ----Iscritto---- 1..1 CorsoDiLaurea)
all X Y (Iscritto(X,Y) -> (Studente(X) & CorsoDiLaurea(Y))).
% Ruolo "Studente"
all X (Studente(X) -> (exists Y Iscritto(X,Y))).                    % {1..*}
all X Y Z ( (Studente(X) & Iscritto(X,Y) & Iscritto(X,Z)) -> Y=Z ). % {0..1}
%  Ruolo "CorsoDiLaurea"
all X (CorsoDiLaurea(X) ->
  (exists Y Y1 (Iscritto(Y,X) & Iscritto(Y1,X) & Y!=Y1))). % {2..*}

% LE CLASSI NON LEGATE DA GERARCHIE SONO DISGIUNTE (un oggetto appartiene ad una sola classe)
all X -(Studente(X) & CorsoDiLaurea(X)).

% OGNI OGGETTO APPARTIENE AD ALMENO UNA CLASSE
all X (Studente(X) | CorsoDiLaurea(X)).

% CONSEGUENZE LOGICHE DA DIMOSTRARE:

% Esiste esattamente uno studente
esattamenteUnoStudente <->
      ((exists X Studente(X)) &
          (all X1 X2 ((Studente(X1) & Studente(X2)) -> X1 = X2))).

%% TESI
-(
   %% I vincoli impediscono che possa esistere un solo studente
   -esattamenteUnoStudente
).
end_of_list.
```

◦

Per quanto riguarda l'inconsistenza finita, purtroppo non è possibile usare Otter o Mace per verificarla (cfr. anche esercizio 7.13). Un metodo, basato sulla programmazione a vincoli è presentato in [7, 6].

7.4.2.2 Verifica di proprietà tramite un generatore di modelli

Il paragrafo precedente ci ha mostrato che è possibile usare in maniera proficua sia un dimostratore di teoremi (OTTER) sia un generatore di modelli (MACE) per ottenere interessanti indicazioni sulle proprietà di un diagramma UML delle classi.

Esistono generatori di modelli progettati con particolare attenzione ai linguaggi di specifica orientati agli oggetti. Un esempio interessante è il sistema ALLOY, disponibile alla pagina `http://alloy.mit.edu`, che ha sia una sintassi che facilita la rappresentazione di classi, associazioni e vincoli, sia efficaci strumenti grafici per lo sviluppo.

La rappresentazione in ALLOY di diagrammi UML delle classi privi di attributi e operazioni segue il seguente schema:

1. codifica del diagramma UML delle classi;
   (a) classi e associazioni;
   (b) identità delle associazioni inverse;
   (c) vincoli {*disjoint*} e/o {*complete*};
2. codifica di vincoli (non rappresentati graficamente);
3. verifica di asserzioni.

La rappresentazione in ALLOY del diagramma di figura 7.19 (ignorando attributi e operazioni), è riportata di seguito.

```
// File visite.als
// Time-stamp: "2006-06-04 22:29:11 cadoli"
// Syntax: Alloy 3

module alloy/visite
// 1.a
sig Persona {
    assistito: set Visita,
    medico: set Visita
}
sig Visita {
    assistito: one Persona,
    medico: one Persona,
    indicazione: set Prescrizione
}
sig Prescrizione {
    indicazione: one Visita
}
// 1.b
fact reverseAssistito {
  all p: Persona |
      p.assistito.assistito = p
}
fact reverseAssistitob {
  all v: Visita |
      v.assistito.assistito = v
}
fact reverseMedico {
  all p: Persona |
      p.medico.medico = p
}
fact reverseMedicob {
  all v: Visita |
      v.medico.medico = v
}
fact reverseIndicazione {
  all p: Prescrizione |
      p.indicazione.indicazione = p
}
fact reverseIndicazioneb {
  all v: Visita |
```

```
      v.indicazione.indicazione = v
}
// 1.c
sig Farmaco, RichiestaRicovero extends Prescrizione { }
fact FRRPartition { Farmaco + RichiestaRicovero = Prescrizione }
// 2.
fact noMedicoAssistito {
// un medico non può mai visitare se stesso
   all v: Visita |
       v.medico != v.assistito
}
// 3.
// trivially false assertion; used to show example
assert show { no Visita }
check show for 2

assert noDueVisiteStessaPrescrizione {
// true assertion: implied by cardinality constraints
// due visite distinte non possono avere prescrizioni in comune
  all v1, v2: Visita |
    v1 != v2 =>
    no v1.indicazione & v2.indicazione
}
check noDueVisiteStessaPrescrizione for 10

// true assertion
assert ogniFarmacoUnMedico {
  all f: Farmaco | one f.indicazione.medico
}
check ogniFarmacoUnMedico for 3

// false assertion
assert ogniMedicoIndicaAlmenoUnFarmaco {
  all p:Persona |
    // se è un medico
    some p.medico =>
        some p.medico.indicazione &&
             p.medico.indicazione = Farmaco
}
check ogniMedicoIndicaAlmenoUnFarmaco for 2

// false assertion
assert unMedicoNonPuoAvereAssistenza {
  all p:Persona |
    some p.medico =>
        no p.assistito
}
check unMedicoNonPuoAvereAssistenza for 2
// NOTA: richiede almeno due visite, quindi non trova controesempi "for 1"
```

Qualche commento sul codice è d'obbligo.

- Nella parte 1.a viene definita una "segnatura" (`sig`) per ogni classe UML e le associazioni in cui è coinvolta. Alcuni vincoli di molteplicità possono essere specificati in forma abbreviata (`set` = "0..*"; `some` = "1..*"; `one` = "1..1"; `lone` = "0..1").
- Nella parte 1.b viene definito come vincolo (`fact`) il fatto che le associazioni di due classi diverse ma con lo stesso nome si riferiscono allo stesso insieme di tuple. Per ogni associazione sono necessari due `fact`.
- Nella parte 1.c la segnatura delle sottoclassi contiene l'indicazione che si ha a che fare con una relazione ISA (`extends`); un vincolo (`fact`) è utile per segnalare che la gerarchia è una partizione.
- Nella parte 2 sono inseriti tutti i vincoli che non hanno controparte grafica in UML. Ad esempio, il vincolo che un medico non possa mai visitare se stesso, espresso in logica tramite la formula

$$\forall XYZ\ Medico(X,Y) \wedge Assistito(Z,Y) \rightarrow X \neq Z,$$

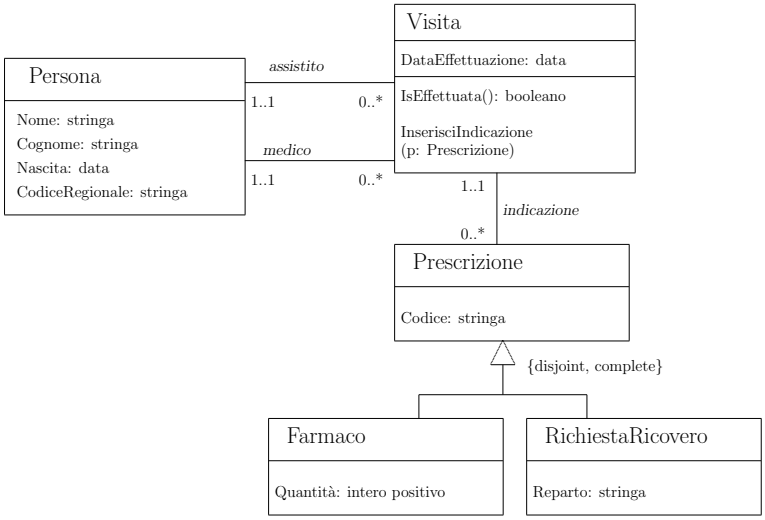

Figura 7.19 Un diagramma UML delle classi

viene specificato come `fact`.

- Nella parte 3 sono inserite asserzioni (`assert`) che si ritengono essere vere o false. ALLOY traduce queste asserzioni nel formato proposizionale DIMACS (cfr. paragrafo 3.4.3) tenendo conto del numero di oggetti richiesti nella parte `check`, che può essere diverso per ogni classe.

  Se l'asserzione risulta vera in tutti i modelli della taglia richiesta, il sistema lo segnala. Ad esempio, nel caso dell'asserzione `noDueVisiteStessaPrescrizione`, che in logica possiamo scrivere come

  $$\forall XYZW\ Indicazione(X,Y) \wedge Indicazione(Z,W) \wedge X \neq Z \rightarrow Y \neq W,$$

  il sistema risponde con il messaggio "`No counterexample found: noDueVisiteStessaPrescrizione may be valid`".

  Se l'asserzione risulta falsa in almeno un modello, il sistema lo fornisce in output in forma grafica. Ad esempio, nel caso dell'asserzione `unMedicoNonPuoAvereAssistenza`, che in logica scriveremmo in questa maniera:

  $$\forall X\ (\exists Y\ Medico(X,Y) \rightarrow \neg\exists Z\ Assistito(X,Z)),$$

  il sistema fornisce come output un grafo in cui i nodi sono gli oggetti e gli archi i link delle associazioni.

Concludendo, possiamo affermare che ALLOY offre alcuni vantaggi rispetto a MACE:

- ha una sintassi per certi versi più adatta alla specifica di diagrammi UML delle classi;
- è più espressivo; ad esempio ha costrutti per la *chiusura transitiva*, che non sono presenti in MACE;
- fornisce la possibilità di scegliere fra vari solutori SAT, fra cui `zchaff`, menzionato nel paragrafo 3.5;
- ha interessanti strumenti per il *reverse engineering*, che forniscono, ad esempio, la possiblità di visualizzare un diagramma (simile a quello delle classi UML) corrispondente al file di testo presentato.

7.5 Esercitazione pratica

7.5.1 Prima parte

Completare gli esercizi del paragrafo 7.3, utilizzando OTTER, MACE e ALLOY.

7.5.2 Seconda parte

Come noto molti sistemi di ascensori (presenti in edifici pubblici o privati, ad esempio negli alberghi) sono costituiti da n cabine che permettono di accedere a p piani ed hanno una gestione centralizzata delle chiamate. Ad ogni piano è possibile premere un tasto per chiedere di andare su o giù, di fatto lasciando al sistema la decisione di quale fra le n cabine utilizzare per soddisfare tale richiesta. Una volta dentro la cabina è possibile premere altri tasti per prenotare la fermata a vari fra i p piani. Il sistema gestisce le richieste "interne" (di cabina) compatibilmente con le richieste "esterne" (di piano). Mentre queste attività vengono svolte il sistema informa gli utenti sull'andamento dello stato, ad esempio accendendo luci e/o attivando suoni nelle cabine e ai piani.

La formalizzazione degli aspetti statici di questo scenario richiede la rappresentazione di vari concetti e relazioni. Un elenco non esaustivo è il seguente:

- piano (compreso primo e ultimo),
- cabina dell'ascensore,
- sensore di presenza della cabina al piano,
- posizione della cabina (al piano o fra due piani),
- pulsantiera (interna alla cabina o situata ad un piano),
- tasto di una pulsantiera (prenota corsa "in su" o "in giù", vai a piano x, ...),
- segnalazione interna o esterna (l'ascensore si trova al piano x, sta andando nella direzione y ed ha prenotazioni per l'insieme P di piani),
- luce di un tasto o di una segnalazione,
- suono da emettere in una certa circostanza (ad es., arrivo a destinazione),
- porta di un ascensore (si potrebbe uscire da un lato diverso rispetto a quello in cui si è entrati).

Ovviamente esistono vari vincoli che la rappresentazione deve soddisfare. Un elenco non esaustivo è il seguente:

- le pulsantiere dei piani hanno al più un tasto di prenotazione corsa "in giù" o "in su",
- la pulsantiera del piano più basso non ha il tasto di prenotazione corsa "in giù",
- le pulsantiere interne alle cabine non hanno il tasto di prenotazione corsa "in giù" o "in su".

L'esercitazione consiste nelle seguenti attività:

1. Rappresentare il maggior numero di requisiti presenti nel tariffario tramite un diagramma UML delle classi, corredato da opportuni vincoli.
2. Tradurre in logica del prim'ordine il diagramma secondo la metodologia illustrata nel paragrafo 7.2.
3. Decidere alcune proprietà che si vogliono verificare, prendendo come esempi quelle elencate nel paragrafo 7.3.
4. Utilizzare OTTER, MACE e ALLOY per la verifica automatica di tali proprietà.

7.6 Nota bibliografica

Formalizzazioni del diagramma delle classi in logica del prim'ordine sono state proposte da vari autori, fra cui [3] (su cui è basata la traduzione presentata in questo capitolo) [17, 40]. Il significato dei vincoli di molteplicità per associazioni non binarie (la cosiddetta semantica *look across*) è riportata in [42]. La dimostrazione di EXPTIME-completezza del ragionamento in diagrammi privi di vincoli è in [3]. Una trattazione della soddisfacibilità finita si può trovare in [10, 9], mentre un'implementazione della tecnica ivi descritta mediante programmazione a vincoli è riportata in [7, 6].

```
relazione ORDINE
 NAT  |  0  |  1  |  2  |  3  |  4  |  5  |
 -----+-----+-----+-----+-----+-----+-----+ .....
 PARI |  0  |  2  |  4  |  6  |  8  | 10  |

relazione APPROSSIMA
 NAT  | 0 1 | 2 3 | 4 5 | 6 7 | 8 9 |10 11|
 -----+-----+-----+-----+-----+-----+-----+ .....
 PARI |  0  |  2  |  4  |  6  |  8  | 10  |
```

Figura 7.20 Due relazioni fra i numeri naturali e numeri i pari

7.7 Esercizi proposti

Esercizio 7.12 [Soluzione a pag. 114] Facendo riferimento all'esercizio 7.10, utilizzare OTTER per trovare la soluzione.

Esercizio 7.13 [Soluzione a pag. 115] Un semplice esempio di formule del prim'ordine che hanno solo modelli infiniti può essere ottenuto considerando le relazioni esistenti fra i numeri naturali e i numeri pari. Infatti ad ogni numero naturale corrisponde biunivocamente un numero pari, secondo la relazione d'ordine. È anche vero che ad ogni numero pari corrispondono due numeri naturali, che sono quelli che, rispettivamente, lo uguagliano o lo approssimano per eccesso. Due presentazioni parziali di tali relazioni sono riportate in figura 7.20. È evidente che tali relazioni non sussistono se si considera un qualsiasi troncamento finito dell'insieme dei numeri, ma sono valide se si considerano tutti gli infiniti numeri naturali.

1. Rappresentare le relazioni apena descritte mediante un diagramma UML delle classi.
2. Rappresentare mediante una formula del prim'ordine tale diagramma e tradurlo nella sintassi di OTTER. Tale formula è soddisfacibile, finitamente insoddisfacibile o insoddisfacibile?
3. Come possiamo usare OTTER e/o MACE per convalidare la risposta data al punto precedente?

7.8 Soluzione di alcuni esercizi

Soluzione esercizio 7.1. Dobbiamo rappresentare il vincolo di insistenza dell'associazione sulle classi e il limite superiore di uno solo dei vincoli di molteplicità (ovvero '3'). In formule, abbiamo:

$$\begin{array}{rl} \forall XY & Esame(X,Y) \rightarrow Studente(X) \wedge Corso(Y), \\ \forall XYZWT & Esame(X,Y) \wedge Esame(X,Z) \wedge Esame(X,W) \wedge Esame(X,T) \rightarrow \\ & Y = Z \ \vee \ Y = W \ \vee \ Y = T \ \vee \ Z = W \ \vee \ Z = T \ \vee \ W = T. \end{array}$$

∘

Soluzione esercizio 7.4. Le formule richieste sono le seguenti:

$$\begin{array}{rl} & // \text{ associazione } \textit{Ordinante} \\ \forall XY & Ordinante(X,Y) \rightarrow Bonifico(X) \wedge ContoCorrente(Y), \\ & // \text{ associazione } \textit{PersonaOrdinante} \\ \forall XY & PersonaOrdinante(X,Y) \rightarrow BonificoDaPersona(X) \wedge IntestatoAPersona(Y), \\ & // \text{ ISA fra classi} \\ \forall X & BonificoDaPersona(X) \rightarrow Bonifico(X), \\ \forall X & IntestatoAPersona(X) \rightarrow ContoCorrente(X), \\ \forall X & IntestatoASocieta(X) \rightarrow ContoCorrente(X), \\ & // \text{ subset fra associazioni} \\ \forall XY & PersonaOrdinante(X,Y) \rightarrow Ordinante(X,Y), \end{array}$$

$$
\begin{array}{rl}
 & // \text{ disjoint di gerarchie} \\
\forall X & Bonifico(X) \rightarrow \neg ContoCorrente(X), // \text{ cfr. fine paragrafo 7.2} \\
\forall X & IntestatoASocieta(X) \rightarrow \neg IntestatoAPersona(X), \\
 & // \text{ complete di gerarchie} \\
\forall X & ContoCorrente(X) \rightarrow IntestatoASocieta(X) \vee IntestatoAPersona(X), \\
 & // \text{ vincoli di molteplicità associazione } Ordinante \\
\forall X & Bonifico(X) \rightarrow \exists Y Ordinante(X, Y), \\
\forall XYZ & Ordinante(X, Y) \wedge Ordinante(X, Z) \rightarrow Y = Z.
\end{array}
$$

○

Soluzione esercizio 7.7. I due diagrammi sono equivalenti in quanto le loro traduzioni in logica del prim'ordine sono formule logicamente equivalenti. Lo si può dimostrare invocando OTTER sul seguente file di input.

```
%%% File: ristrutturazione.ott
%%% Time-stamp: "2006-08-07 10:37:46 cadoli"
%%% Formato: file di input per OTTER 3.3

set(auto).
formula_list(usable).

%% ISA FRA CLASSI
A1 <-> (all X (Maschio(X) -> Persona(X))).
A2 <-> (all X (Femmina(X) -> Persona(X))).

%% COMPLETE DI GERARCHIE
A3 <-> (all X (Persona(X) -> (Maschio(X) | Femmina(X)))).

%% ATTRIBUTO CLASSE Maschio
A4 <-> (all X Y (Maschio(X) & Eta(X,Y) -> Int(Y))).          % TIPO ATTRIBUTO
A5 <-> (all X (Maschio(X) -> (exists Y Eta(X,Y)))).          % ATTRIBUTO {1..*}
A6 <-> (all X Y Z (Maschio(X) & Eta(X,Y) & Eta(X,Z) -> Y=Z)).% ATTRIBUTO {0..1}

%% ATTRIBUTO CLASSE Femmina
A7 <-> (all X Y (Femmina(X) & Eta(X,Y) -> Int(Y))).          % TIPO ATTRIBUTO
A8 <-> (all X (Femmina(X) -> (exists Y Eta(X,Y)))).          % ATTRIBUTO {1..*}
A9 <-> (all X Y Z (Femmina(X) & Eta(X,Y) & Eta(X,Z) -> Y=Z)).% ATTRIBUTO {0..1}

%%%%% MODELLAZIONE POCO COESA: attributo Età nelle sottoclassi
MPC <-> A1 & A2 & A3 & A4 & A5 & A6 & A7 & A8 & A9.

%% ATTRIBUTO CLASSE Persona
A10<-> (all X Y (Persona(X) & Eta(X,Y) -> Int(Y))).          % TIPO ATTRIBUTO
A11<-> (all X (Persona(X) -> (exists Y Eta(X,Y)))).          % ATTRIBUTO {1..*}
A12<-> (all X Y Z (Persona(X) & Eta(X,Y) & Eta(X,Z) -> Y=Z)).% ATTRIBUTO {0..1}

%%%%% MODELLAZIONE COESA: attributo Età nella superclasse
MC <-> A1 & A2 & A3 & A10 & A11 & A12.

%% CONSEGUENZA LOGICA: le due modellazioni sono logicamente equivalenti
-(MPC <-> MC).

end_of_list.
```

○

Soluzione esercizio 7.8. Le formule richieste sono le seguenti:

1. $\forall XY\ BonificoDaPersona(X) \wedge Ordinante(X, Y) \rightarrow IntestatoAPersona(Y)$.
2. $\forall XY\ IntestatoAPersona(X) \wedge Ordinante(Y, X) \rightarrow BonificoDaPersona(Y)$.

Poiché nessuna di esse è una conseguenza logica della formula Φ che traduce il diagramma, la cosa più opportuna, nel caso probabile in cui il committente lo desideri, è aggiungerle come vincoli. ◦

Soluzione esercizio 7.10. Le formule richieste sono le seguenti:

$$\begin{array}{ll}
\forall X & Italian(X) \rightarrow Person(X), \\
\forall X & English(X) \rightarrow Person(X), \\
\forall X & Italian(X) \rightarrow \neg English(X), \\
\forall X & Gentleman(X) \rightarrow English(X), \\
\forall X & Hooligan(X) \rightarrow English(X), \\
\forall X & Lazy(X) \rightarrow Italian(X), \\
\forall X & LatinLover(X) \rightarrow Italian(X), \\
\forall X & Lazy(X) \rightarrow \neg LatinLover(X), \\
\forall X & Italian(X) \rightarrow LatinLover(X) \vee Lazy(X), \\
\forall X & LatinLover(X) \rightarrow Gentleman(X).
\end{array}$$

La classe *LatinLover* è inconsistente, infatti, posto ugule a Φ l'insieme delle regole appena elencato, si ha che:

$$\Phi \models \forall X \ \neg LatinLover(X).$$

Per ristabilire la consistenza è inutile eliminare il vincolo {*disjoint, complete*} fra le classi *Lazy* e *LatinLover*. Risolviamo il problema:

- eliminando il vincolo {*disjoint*} fra le classi *Italian* e *English*, oppure
- eliminando la relazione ISA fra le classi *LatinLover* e *Gentleman*.

◦

Soluzione esercizio 7.12. Il file richiesto è il seguente:

```
%%% File: gentlemen.ott
%%% Time-stamp: "2006-06-04 12:02:44 cadoli"
%%% Formato: file di input per OTTER 3.3

set(auto).
set(display_terms).
formula_list(usable).

all X (Italian(X) -> Person(X)).
all X (English(X) -> Person(X)).
all X (Italian(X) -> -English(X)).                % disjoint

all X (Gentleman(X) -> English(X)).
all X (Hooligan(X) -> English(X)).

all X (Lazy(X) -> Italian(X)).
all X (LatinLover(X) -> Italian(X)).
all X (Lazy(X) -> -LatinLover(X)).                % disjoint
all X (Italian(X) -> LatinLover(X) | Lazy(X)).    % covering

all X (LatinLover(X) -> Gentleman(X)).

%%% Conseguenza logica: non esiste alcun LatinLover
%%% NOTA: la conseguenza vale anche se non c'è l'asserzione {disjoint,covering}
exists X LatinLover(X).

end_of_list.
```

◦

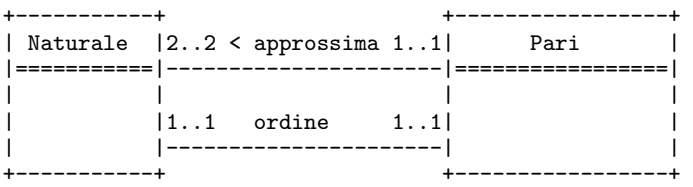

Figura 7.21 Diagramma UML delle classi per le relazioni della figura 7.20 (esercizio 7.13)

Soluzione esercizio 7.13.

1. Il diagramma richiesto è riportato in figura 7.21.
2. La formula richiesta, direttamente nella sintassi di OTTER, è riportata di seguito:

```
%%% File: nat-pari.ott
%%% Time-stamp: "2006-06-04 16:41:24 cadoli"
%%% Formato: file di input per OTTER 3.3

set(auto).
formula_list(usable).
% ASSIOMI GENERALI
% OPZIONALE: LE CLASSI SONO DISGIUNTE, ovvero UN OGGETTO APPARTIENE AD UNA SOLA CLASSE
(all X (-(Nat(X) & Pari(X)))).
% OGNI OGGETTO APPARTIENE AD ALMENO UNA CLASSE
(all X (Nat(X) | Pari(X))).
% TIPIZZAZIONE DELLE ASSOCIAZIONI BINARIE
(all X Y (Appr(X,Y) -> (Nat(X) & Pari(Y)))).
(all X Y (Ordine(X,Y) -> (Nat(X) & Pari(Y)))).

% ASSIOMI DEI VINCOLI DI MOLTEPLICITÀ
%%% Appr
%  Nat -Appr-1..* Pari
all X (Nat(X) -> (exists Y Appr(X,Y))).
%  Nat -Appr-0..1 Pari
all X Y Z (Appr(X,Y) & Appr(X,Z) -> Y = Z).
%  Pari -Appr-2..* Nat
all X (Pari(X) -> (exists Y Y1 (Appr(Y,X) & Appr(Y1,X) & Y != Y1))).
%  Pari -Appr-*..2 Nat
all X Y Y1 Y2 (Appr(Y,X) & Appr(Y1,X) & Appr(Y2,X) ->
                Y = Y1 | Y = Y2 | Y1 = Y2).
%%% Ordine
%  Nat -Ordine-1..* Pari
all X (Nat(X) -> (exists Y Ordine(X,Y))).
%  Nat -Ordine-0..1 Pari
all X Y Z (Ordine(X,Y) & Ordine(X,Z) -> Y = Z).
%  Pari -Ordine-1..* Nat
all X (Pari(X) -> (exists Y Ordine(Y,X))).
%  Pari -Ordine-0..1 Nat
all X Y Y1 (Ordine(Y,X) & Ordine(Y1,X) -> Y = Y1).

end_of_list.
```

   La formula è soddisfacibile, ma entrambe le classi sono finitamente insoddisfacibili.

3. Ovviamente OTTER non riesce a refutare trovando una contraddizione, perché la formula è soddisfacibile. MACE non ce la fa a trovare modelli finiti, perché non ne esistono; in meno di un minuto è possibile verificare questa affermazione con modelli di taglia fino a 12 (opzione `-N`).

○