Capitolo 4

Strutture dati ed algoritmi

In questo capitolo verranno descritte le strutture dati e gli algoritmi sono state definite al fine di fornire il nuovo prototipo, verrà inoltre illustrato il diagramma delle classi implementate in modo tale da avere una visione di tutto il prototipo.

4.1 Strutture dati principali utilizzate per il prototipo

Questo paragrafo vuole essere una sintesi esplicativa e descrittiva di quali sono e di come vengono utilizzate le strutture dati in Penelope per generare fisicamente programmi.

ParserPage e **ParserStructure**

La classe ParserPDL costruisce in memoria una struttura di record e puntatori (ParserPage, ParserStructure) associata ad una singola istruzione DEFINE PAGE incontrata durante la fase di "parsing". Tale struttura viene utilizzata dalle classi Checker, GenerateDDL, EditorTemplate, GenerateML, GenerateSources, PreGenerateML. Va notato che questa struttura viene modificata dalla classe Checker (per poter verificare la correttezza delle definizioni); per questo motivo la fase di check deve essere eseguita una ed una sola volta prima di qualsiasi operazione. La classe ParserPage rappresenta l'istanza della singola pagina definita dall'istruzione DEFINE PAGE; essa contiene una lista di oggetti ParserStructure, uno per ogni attributo definito nella pagina. Poiché gli attributi

ADM possono essere nidificati, un oggetto di tipo ParserStructure può contenere altri oggetti della stessa classe.

TablePage e **TableStructure**

Ad ogni schema di pagina sono associati degli oggetti che conterranno, a tempo d'esecuzione, dei cursori $\texttt{TableR}_h$. Gli oggetti TablePage e TableStructure sono utilizzati al fine di definire opportunamente questi cursori. Un oggetto della classe TablePage definisce un oggetto cursore $\texttt{TableR}_0$ e contiene tanti oggetti di tipo TableStructure quanti sono gli attributi definiti nella pagina. Un oggetto TableStructure può essere vuoto (se associato ad un attributo semplice) o definire un oggetto cursore $\texttt{TableR}_h$ (se associato ad un attributo composto). Queste strutture sono costruite dalla classe PreGenerateML e sono successivamente passate alla classe GenerateSources e GenerateASP.

PreGenerateML

La generazione della pagina associata ad un singolo page-scheme richiede una serie di controlli approfonditi e di fasi per la generazione di strutture intermedie. Queste operazioni sono esplicate una ed una sola volta (prima di procedere alla fase di generazione vera e propria) in modo da ridurre i costi di computazione. La classe PreGenerateML sulla base delle strutture ParserPage e ParserStructure genera oggetti della classi TablePage e TableStructure, oggetti che vengono utilizzati per la definizione dei cursori $\texttt{TableR}_h$.

InstancePage

La classe GenerateSources e GenerateASP costruisce in memoria una struttura di record e puntatori (InstancePage, InstanceStructure) associata alla pagina da generare; ad ogni interazione dell'algoritmo di generazione tale struttura viene materializzata sul file system. La classe InstancePage rappresenta l'istanza della singola pagina; essa contiene una lista di oggetti InstanceStructure, uno per

ogni attributo ADM definito nella pagina. Poiché gli attributi ADM possono essere nidificati, un oggetto di tipo InstanceStructure può contenere altri oggetti della stessa classe.

PenelopeDynamicTable

Un oggetto della classe PenelopeDynamicTable è una query che viene inserita nel file di output utilizzato per generare sia pagine statiche sia pagine dinamiche.
Supponiamo di avere il seguente Page-Scheme:

DEFINE PAGE PageName
AS URL(U_1, U_2,..,U_i)
A_1;
A_2;
Aj;
USING R_1,R_2, ..,R_k
WHERE C_1, C_2,.. C_n
ORDER BY O_1, O_2, .., O_m

Ad esso possiamo associare una serie di oggetti PenelopeDynamicTable così definiti:

- "Query" è la stringa contenente la query che, quando eseguita a tempo dinamico[1], permette di estrarre gli attributi di livello 0;
- "QueryforList" è la stringa contenente la query che, quando eseguita a tempo dinamico, permette di estrarre gli attributi di livello *i* associati ad ogni attributo A_h di tipo LIST-OF.

Poiché gli attributi A_h possono essere nidificati, tali nidificazioni danno luogo ad altre query definite in maniera analoga.

[1] A tempo dinamico si intende all'atto della visualizzazione della pagina web richiesta da un client; a tempo statico si intende all'atto di generazione fisica della pagina attraverso Penelope

Di seguito è schematizzata la struttura ad albero che viene creata in memoria dai moduli precedentemente descritti.

ESEMPIO

Supponiamo di aver definito il seguente Page-Scheme :

```
DEFINE PAGE PageName
AS URL(U1,..,UN);
   Attributo1: TEXT = Valore1;
   Attributo2: TEXT = Valore2;
   Attributo3: LIST-OF (
                        Attributo31: TEXT = Valore31;
                        Attributo32: TEXT = Valore32;
                        Attributo33: TEXT = Valore33;
                       );
   USING Tabella1,..,TabellaN
   END
```

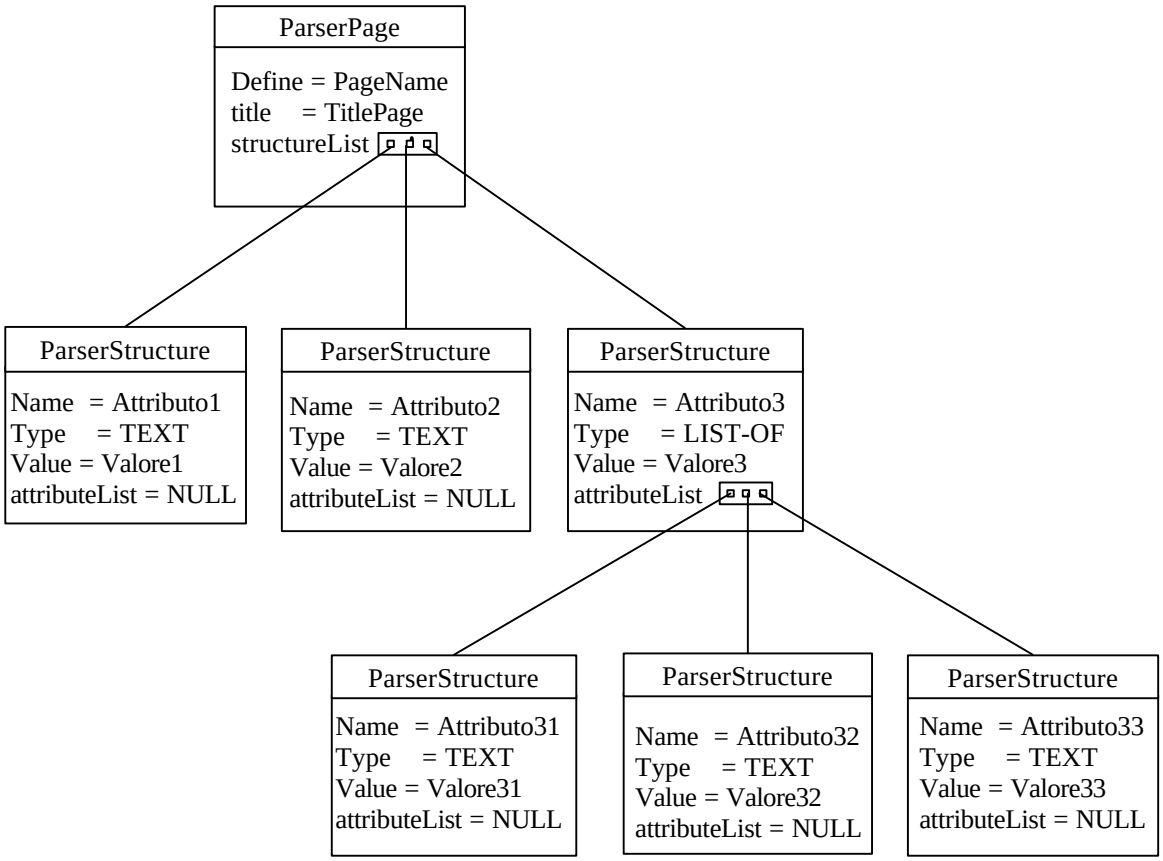

Figura 4.1: Struttura generata in memoria dal modulo “Parser”

Come può essere dedotto dalla figura per ogni page-scheme del PL abbiamo un oggetto istanza di `ParserPage`. All’interno di ogni oggetto `ParserPage` sono presenti un campo *define* (stringa che rappresenta il nome del page-scheme) e una lista (un vettore) chiamata *structureList*, dove ogni item è un riferimento ad un oggetto che rappresenta un attributo di livello 0 del page-scheme; ognuno di questi oggetti è un’istanza della classe `ParserStructure`.

Nella *ParserStructure* sono presenti il nome dell’attributo (campo *name*), il tipo dell’attributo (campo *type* che può assumere valori TEXT, IMAGE, LIST-OF, LINK-TO, MAP, FORM, etc..) e un vettore *attributeList* attraverso il quale, se l’attributo è composto, è possibile accedere agli altri oggetti di tipo `ParserStructure` associati agli attributi contenuto nell’attributo in esame.

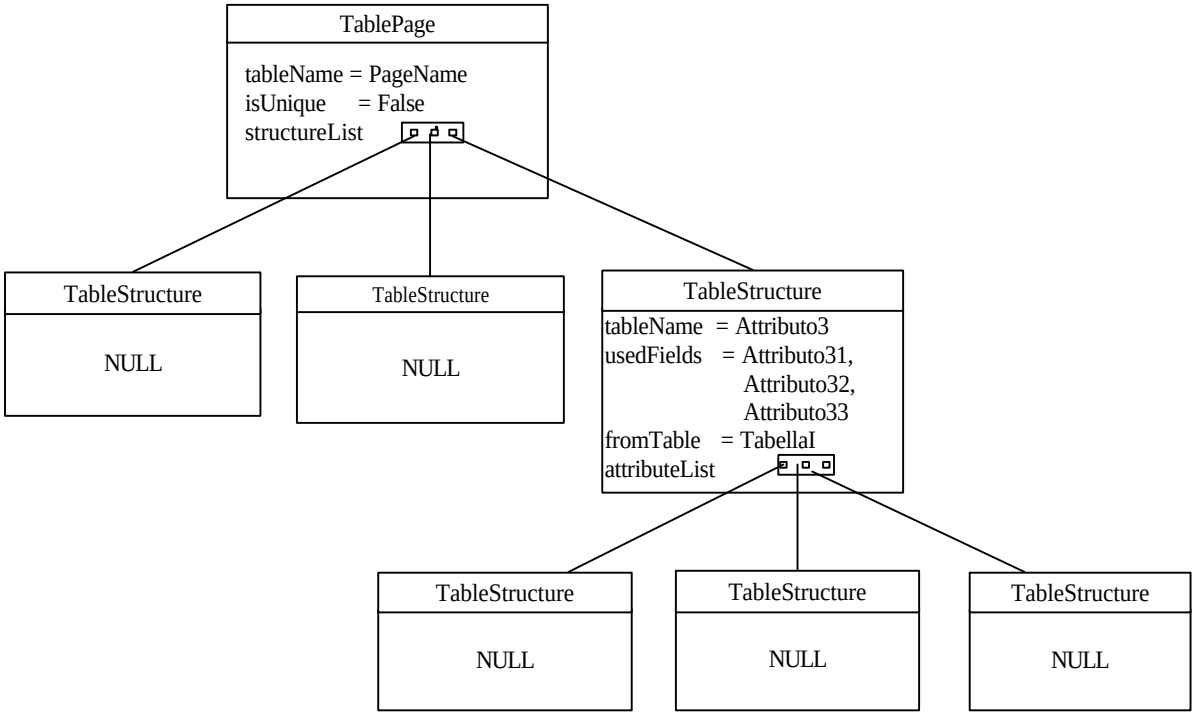

Figura 4.2: Struttura generata in memoria dal modulo "PreGenerate"

Nella TablePage sono presenti le informazioni per creare le query che permettono di istanziare gli attributi semplici di livello 0. Nella struttura è presente un vettore *structureList* che contiene tanti elementi quanti sono gli attributi di livello 0 (cioè quanti sono gli aggetti di tipo ParserStructure). Tali elementi sono di tipo TableStructure e contengono il nome dell'attributo (*tableName*), i campi utilizzati (*usedFields*) e da quali tabelle gli stessi possono essere estratti (*fromTable*).

Nel caso si abbia a che fare con un attributo semplice, l'oggetto TableStructure corrispondente sarà "NULL".

In sintesi è stato descritto come vengono create in memoria due strutture ad albero tra loro sovrapponibili, infatti dalla prima struttura descritta (Figura 4.1) si hanno informazioni sugli attributi mentre nella seconda (Figura 4.2) si hanno informazioni su come, nel caso di attributo composto, generare le query che permettono di estrarre gli attributi.

4.2 Il concetto di *tabella dominante*

Come già accennato, i valori degli attributi del generico livello *i* possono essere costanti oppure provenire dalla base di dati, possiamo pertanto associare ad ogni livello una o più tabelle dalle quali vengono estratti tali valori.

Poiché la struttura memorizzata non è altro che un albero n-ario possiamo addurre una definizione ricorsiva:

- La tabella dalla quale vengono estratti tutti gli attributi di livello 0 viene chiamata "*tabella dominante di livello 0*".
- Per ogni attributo composto di livello *i* indichiamo con il termine "*tabella dominante di livello i*" la vista definita per quel livello. Appare evidente che avremo non una tabella ma un insieme di tabelle dalle quali possono essere estratti i valori degli attributi semplici per quel livello. In particolare tale insieme sarà costituito dalla tabella dominante per quel livello più le tabelle dominanti dei livelli precedenti incontrate nel percorso a ritroso dell'albero.

Per comprendere meglio il concetto forniamo di seguito un esempio:

ESEMPIO

Supponiamo di aver definito il seguente Page-Scheme:

```
DEFINE PAGE PageName
AS URL(<nome>);
Attributo01: AttributoComposto (
        Attributo111: AttributoSemplice = Valore;
            Attributo112: AttributoSemplice = Valore;
            Attributo113: AttributoComposto (
                                Attributo21131: AttributoSemplice = Valore;
                            Attributo21132: AttributoSemplice = Valore;
                                        );
                        );
Attributo02: AttributoSemplice = Valore;
```

$Attributo_0$3: AttributoComposto (

$Attributo_1$31: AttributoSemplice = Valore;

$Attributo_1$32: AttributoSemplice = Valore;

$Attributo_1$33: AttributoComposto (

$Attributo_2$331: AttributoSemplice = Valore;

$Attributo_2$332: AttributoSemplice = Valore;

);

);

USING R0, R1, R2, R4

END

ove AttributoSemplice = {TEXT, IMAGE, LINK-TO}

AttributoComposto = {LIST-OF, FORM, MAP}

Il pedice sugli attributi indica il loro livello di nidificazione.

Supponiamo inoltre che :

R0 sia la tabella dominante per il livello 0;

R1 sia la tabella dominante per l'attributo $Attributo_0$1 (attributo di livello 0 composto);

R2 sia la tabella dominante per l'attributo $Attributo_1$13; (attributo di livello 1 composto);

R3 sia la tabella dominante per l'attributo $Attributo_0$3; (attributo di livello 0 composto);

R4 sia la tabella dominante per l'attributo $Attributo_1$33; (attributo di livello 1 composto);

Quindi possiamo dire che :

- i valori non costanti degli attributi semplici dell'$Attributo_0$1 possono essere estratti sia da R0 che da R1;
- i valori non costanti degli attributi semplici dell'$Attributo_1$13 possono essere estratti da R0, R1, R2;
- i valori non costanti degli attributi semplici dell'$Attributo_0$3 possono essere estratti sia da R0 che da R3;
- i valori non costanti degli attributi semplici dell'$Attributo_1$33 possono essere estratti da R0, R3, R4.

Graficamente possiamo rappresentare quanto detto nel seguente modo:

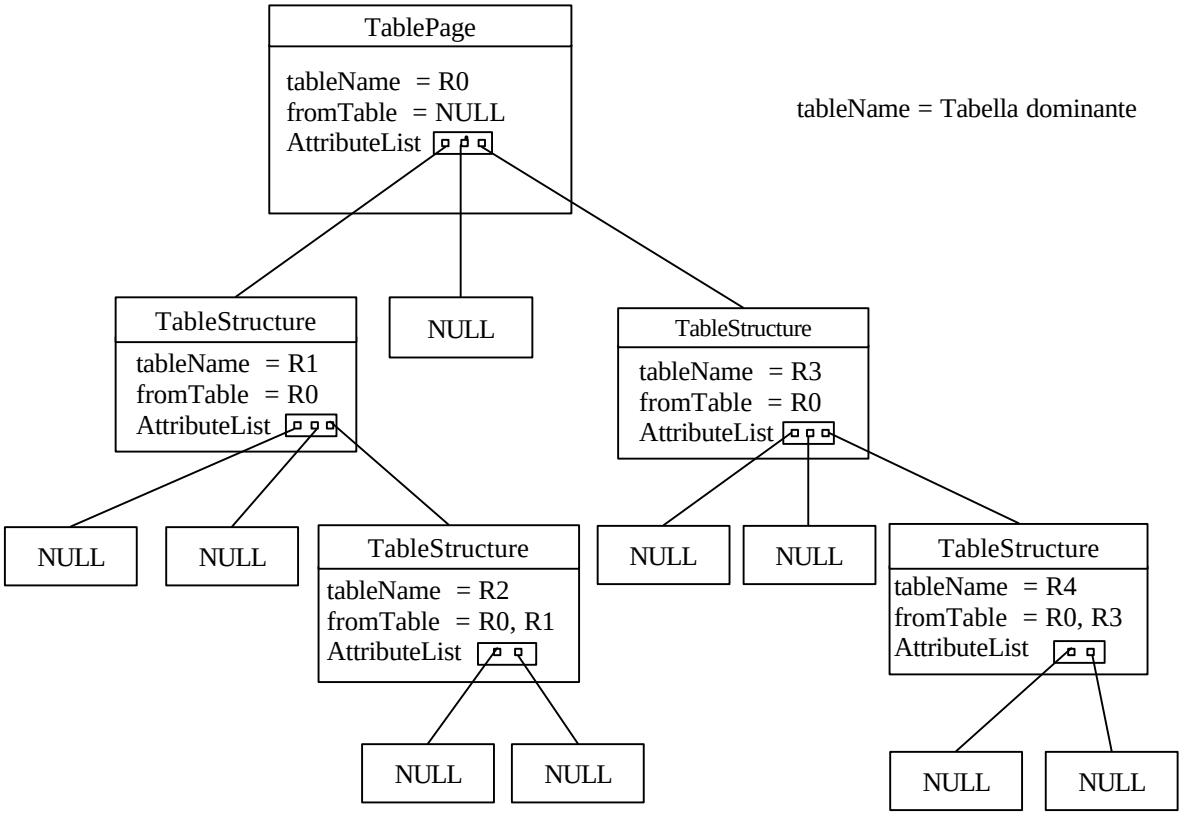

Figura 4.3: Rappresentazione grafica del concetto di *tabella dominante*

4.3 Diagramma delle principali classi implementate

Prima di entrare nel dettaglio degli algoritmi di generazione illustriamo come è organizzato l'intero prototipo Penelope dal punto di vista delle classi java.

In figura 4.4 viene riportata una schematizzazione delle principali classi implementate per il prototipo, come già evidenziato nel paragrafo 3.2 la classe PenelopePDL svolge il ruolo di modulo guida, cioè è da lei che vengono richiamate le classi principali.

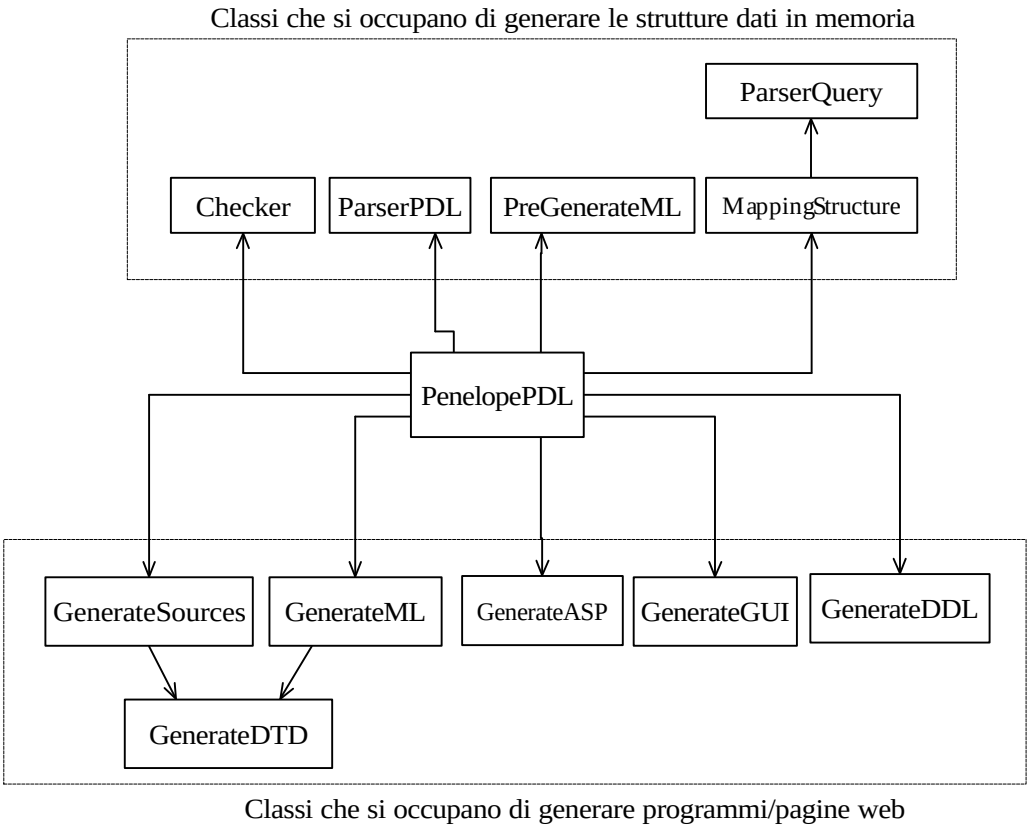

Figura 4.4: Diagramma delle classi di alto livello del prototipo Penelope

Si osservi che vengono distinte due tipologie di classi:

1. classi adibite alla generazione delle strutture in memoria;
2. classi adibite alla generazione programmi o di pagine web.

Vediamo il ruolo svolto da ogni singola classe appartenente alla seconda tipologia:

- la classe `GenerateSources` genera programmi Java e pagine JSP;
- la classe `GenerateML` genera direttamente pagine HTML o XML;
- la classe `GenerateASP` genera pagine ASP;
- la classe `GenerateGUI` genera, nel caso di generazione di siti statici o misti, una interfaccia grafica che permette all'utente, in maniera molto semplice, di selezionare i page-scheme statici (per i quali è stato generato un opportuno

programma Java) e di generare, compilando ed eseguendo i sorgenti Java, pagine HTML o XML;

- nel caso si vogliano generare pagine in formato XML, la classe GenerateDTD crea il DTD (Document Type Definition).

Per quanto riguarda le classi GenerateSources e GenerateML poiché sono due classi fondamentali possiamo ulteriormente espandere il diagramma delle classi nel seguente modo:

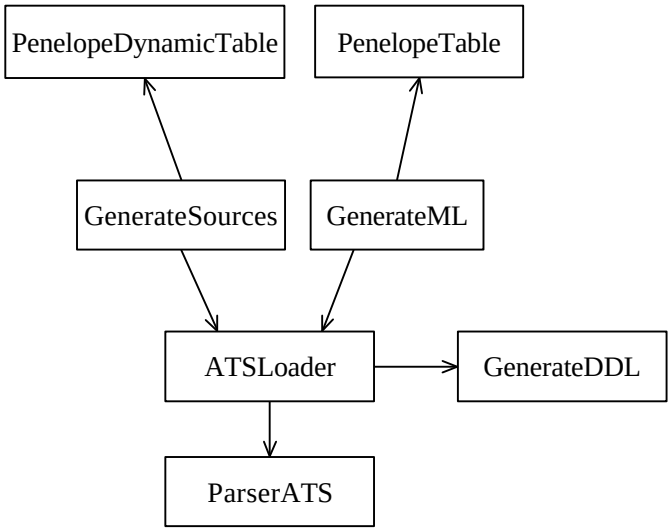

Figura 4.5: Diagramma delle classi di dettaglio per la generazione di programmi o pagine web

- la classe PenelopeTable, utilizzata per la generazione diretta di pagine Web, si occupa di costruire la varie query e di eseguirle;
- la classe PenelopeDynamicTable, utilizzata per la generazione di programmi e di pagine dinamiche, si occupa di costruire la varie query e restituirle come stringhe;
- la classe GenerateDDL genera un file con la rappresentazione ADM del sito;
- le classi ATSLoader e ParserATS caricano in memoria le strutture utilizzate per la realizzazione del layout grafico.

Di seguito riportiamo per ogni classe descritta la sua schematizzazione UML:

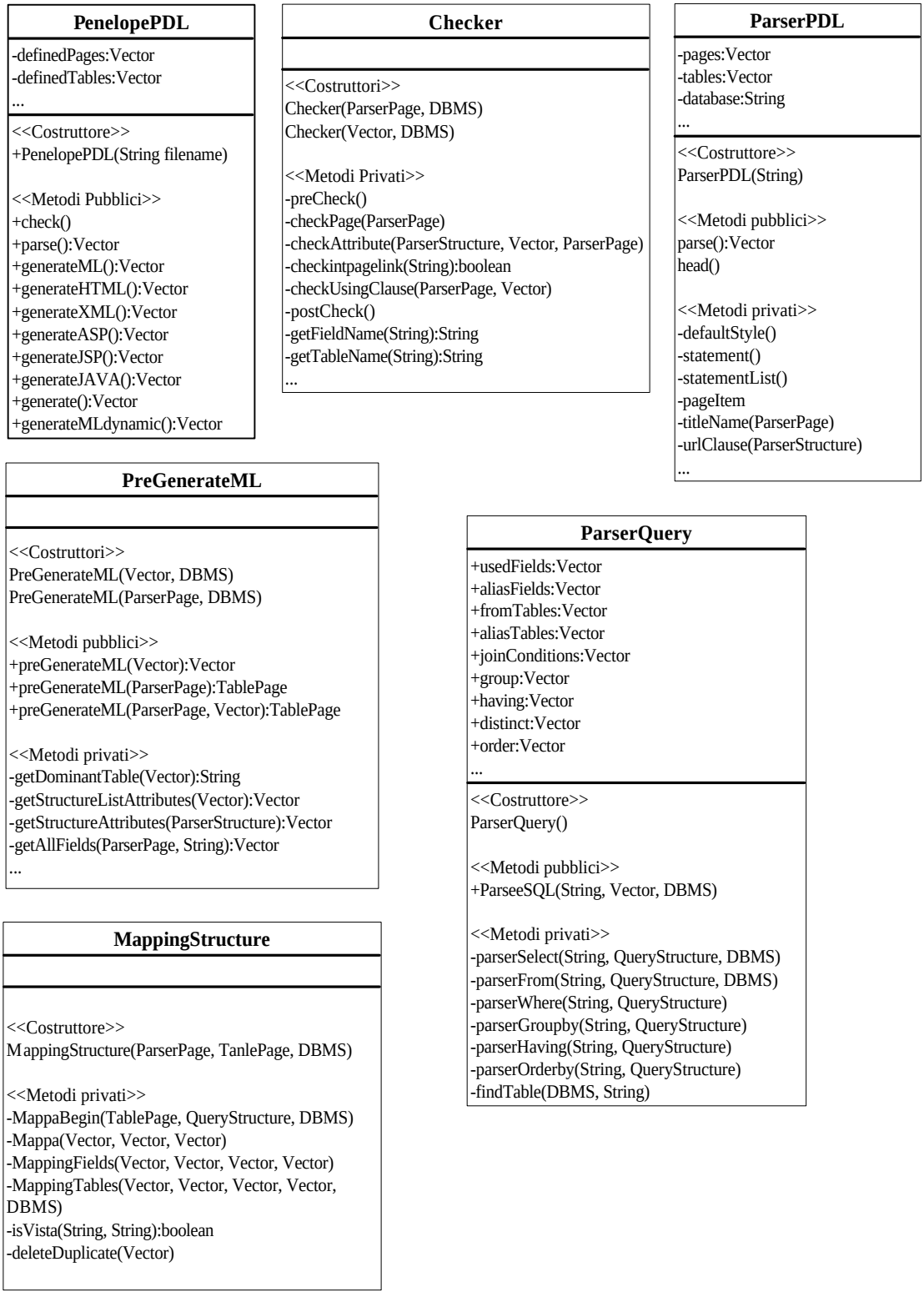

Figura 4.6: Schematizzazione UML delle classi adibite alla generazione delle strutture in memoria

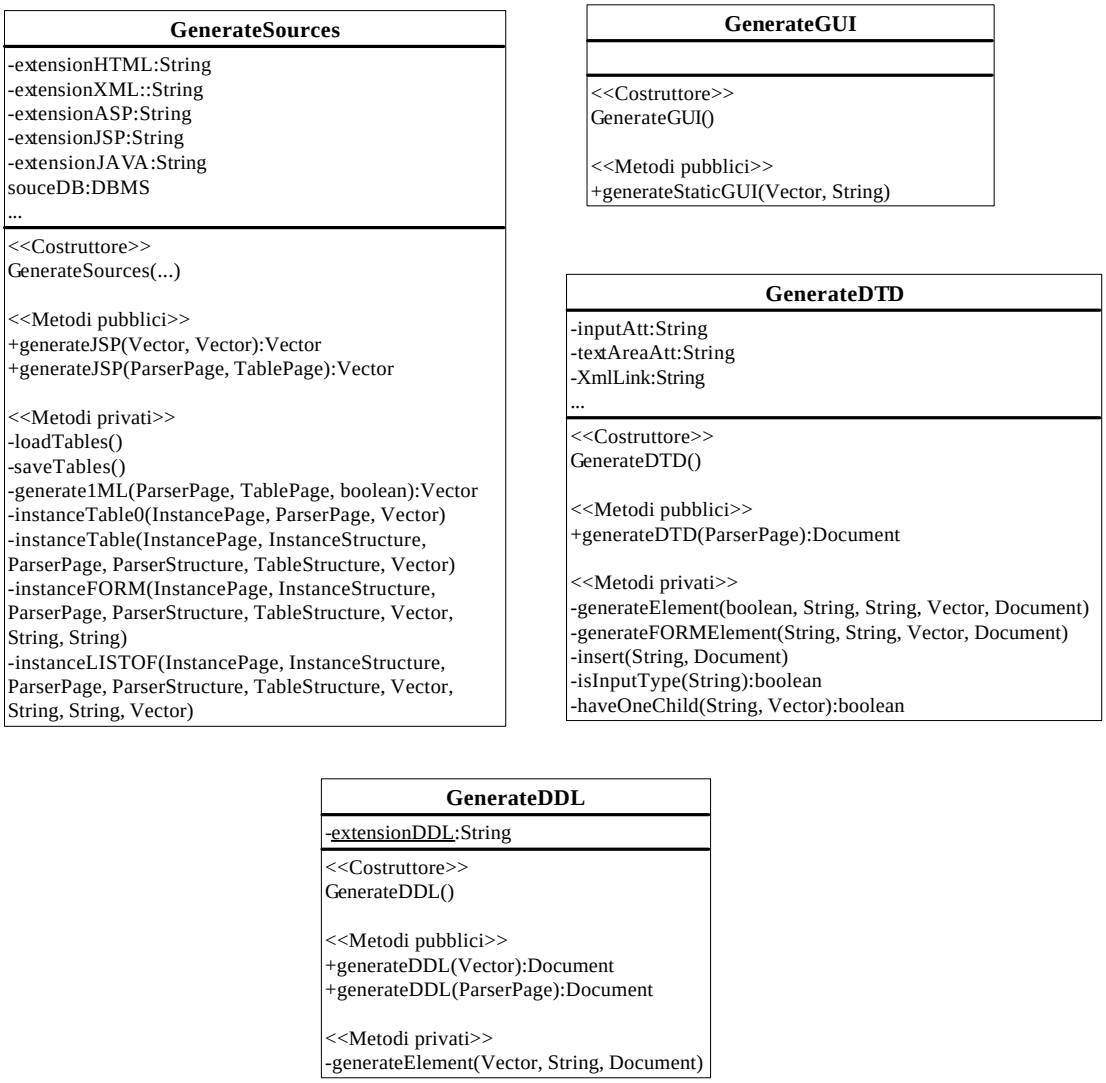

Figura 4.7: Schematizzazione UML delle classi adibite alla generazione di programmi

4.4 Algoritmo di generazione delle pagine

In questo paragrafo verrà descritto in maniera dettagliata l'algoritmo di generazione di pagine dinamiche il quale però richiede degli ulteriori algoritmi che permettano una efficiente gestione delle query definite nella clausola USING di Penelope e che vengono illustrati nel seguito.

Consideriamo uno schema di pagina del tipo :

DEFINE PAGE PageName
AS URL(U1, U2,..,Ui)
A1;
A2;
Aj;
USING R_1,R_2, ..,R_k
WHERE C_1, C_2,.. C_n
ORDER BY O_1, O_2, .., O_m

In questo modo viene definito un page-scheme PageName; l'URL di ogni pagina, istanza dello schema, è costruito sulla base degli attributi[2] U_1, U_2, .. , U_i. Nella pagina sono presenti attributi A_1, A_2, .. , A_j i cui valori possono essere costanti, oppure provenire dalle viste[3] R_1, R_2, .. , R_k. I dati presenti nelle suddette relazioni sono preventivamente filtrati ed ordinati utilizzando le clausole C_1, C_2, .., C_n e O_1, O_2, .., O_m

A questo page-scheme associamo:

Una vista di partenza T_0. Questa coincide con una delle viste R_i e contiene almeno tutti i campi utilizzati dalla funzione URL().

Va notato che :

1. ogni campo utilizzato dalla funzione URL() deve necessariamente o provenire dalla stessa vista T_0 o assumere valore costante;

[2] Nei paragrafi che seguono intenderemo sempre per "attributo" un elemento definito all'interno di un page-scheme e per "campo" il generico attributo di una tabella (o vista) definita in una base di dati.
[3] Inoltre per "vista" intenderemo una tabella o una vista relazionale definita nella base di dati.

2. sarà generata un'unica pagina parametrica per ogni tupla di valori della funzione URL;
3. ogni campo associato ad un attributo semplici A_i deve necessariamente provenire dalla stessa vista T_0 o assumere valore costante;
4. nel caso in cui la funzione URL() abbia argomenti costanti, la vista T_0 coincide con la tabella da cui provengono i campi degli attributi semplice A_i.
5. esiste una tabella dominante T_h per ogni attributo composto (LIST-OF) A_h. Poiché gli attributi A_h possono contenere altri attributi composti A_{hk}, potranno essere definite ricorsivamente tabelle dominanti T_{hk} per ciascuno dei suddetti attributi.

Al fine di agevolare la comprensione di quanto detto si osservi la figura 4.8; il page-scheme "PageName" è costituito da n attributi, alcuni semplici (A_1, A_2,…) ed altri composti (A_h,…). All'interno degli attributi composti, come A_h, sono definiti altri attributi semplici ed altri attributi composti, quali $A_{h..k}$.
Per ogni nodo (rappresentante un attributo composto) dell'albero raffigurato viene riportata la tabella dominante.

6. ogni attributo semplici A_{hj} (attributi definiti in A_h) deve necessariamente provenire dalla stessa vista T_h, o da tabelle dominanti del livello precedente (T_0), od ancora assumere valore costante;
7. analogamente ogni gli attributi semplice A_{hki} (attributo definito nell'attributo composto A_{hk}) deve necessariamente provenire dalla stessa vista T_{hk}, o dalle tabelle dominanti del livello precedente (T_0, T_h) od ancora assumere valore costante; e così via ricorsivamente.

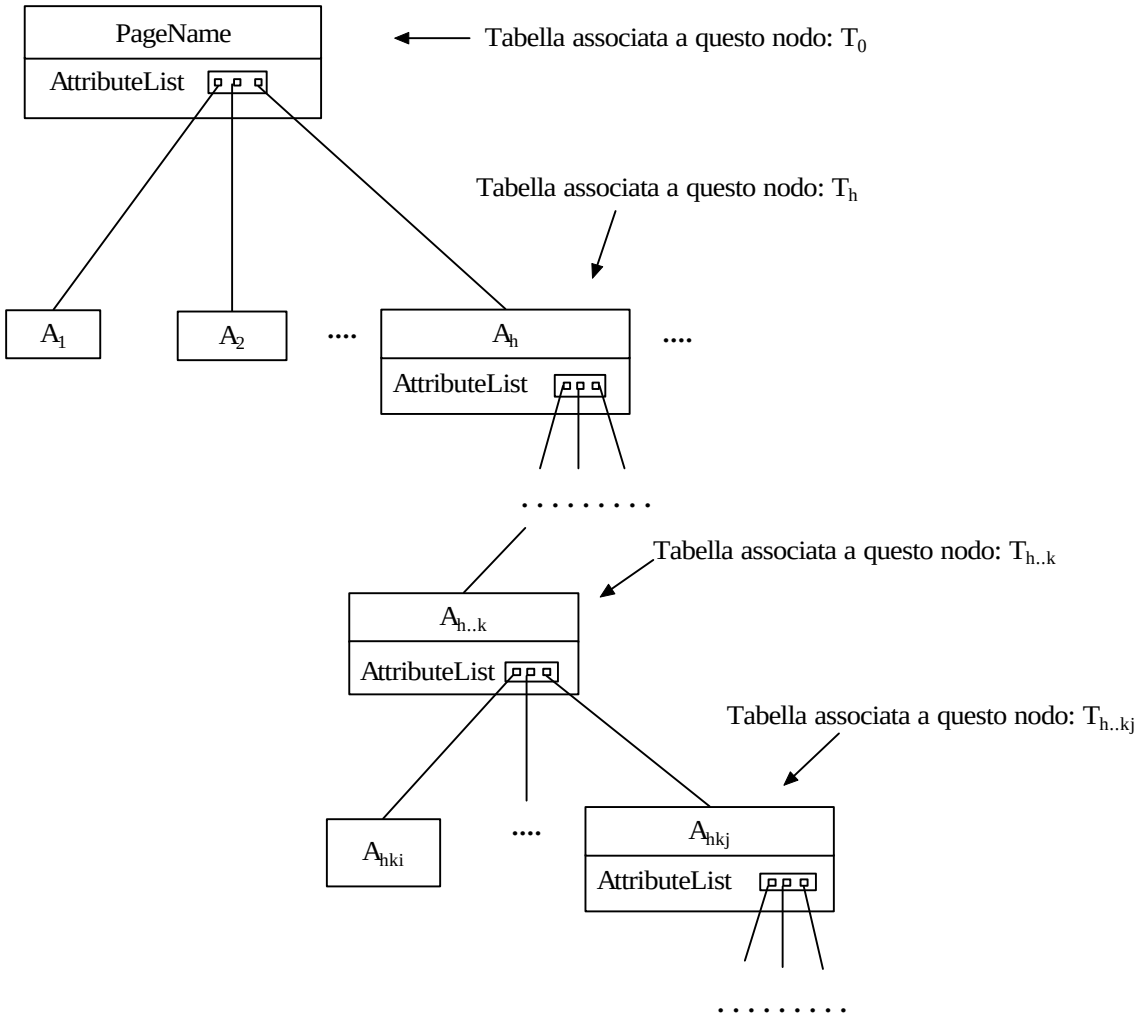

Figura 4.8: Tabelle dominanti per gli attributi composti

L'algoritmo di generazione può essere così descritto:

- definiamo un oggetto Table_T_0 che conterrà un cursore definito a partire dalla vista T_0 a cui sono state eventualmente applicate le selezioni $\{C_i\}$;

$$\text{Table\_}T_0 = \sigma_{C1,\,C2,\,..,\,Cn}(T_0)$$

- indichiamo con $T_0.F_j$ il campo F_j della vista T_0;
- indichiamo con table_$t_0.f_j$ il valore assunto dal campo F_j nell'ennupla corrente del cursore Table_T_0;

- per ogni Page-Scheme
    1. **S**={};
    2. generiamo il nome del file a partire dai valori della funzione URL
    3. per ogni attributo U_i, non costante, aggiungiamo all'insieme delle selezioni **S** la condizione $T_0.U_i$ = table_$t_0.u_i$
    4. istanziamo gli attributi di livello 0;
        4.1 istanziamo il titolo;
        4.2 istanziamo (creando in memoria oggetti di tipo InstanceStructure) ogni attributo semplice A_i; nel caso in cui l'attributo assuma un valore proveniente dal campo F_j, aggiungiamo la condizione $T_0.F_j$ = table_$t_0.f_j$ all'insieme **S**;
    5. istanziamo ogni attributo composto A_h eseguendo la funzione: InstanceTable (A_h, **S**, T_0)
    6. Salviamo sul file system la rappresentazione costruita in memoria ed aggiorniamo la tavola dei file e dei simboli.

La funzione InstanceTable ($A_{h..ki}$, **S**, $T_{h..k}$) è una funzione ricorsiva che prende in ingresso l'attributo composto $A_{h..ki}$, l'insieme **S** delle selezioni, e la tabella dominante $T_{h..k}$ incontrata al passo precedente, e può essere così descritta:

- definiamo un oggetto Table_$T_{h..ki}$ che conterrà il cursore associato a questa iterazione. Tale cursore sarà costruito a partire dalla tabella dominante $T_{h..ki}$ e da quella individuata nel livello precedente ($T_{h..k}$) nel seguente modo :

$$\text{Table\_}T_{h..ki} = \sigma_s(\sigma_{C1, C2, .., Cn} (T_{h..k} \triangleright\triangleleft T_{h..ki}))$$

  Dove $\triangleright\triangleleft$ è da considerarsi come un join naturale;

- Per ogni ennupla del cursore Table_$T_{h..ki}$
    - $\mathbf{S}_n$ ={};
    - istanziamo ogni attributo semplice $A_{h..kij}$, definito nell'attributo composto $A_{h..ki}$; nel caso in cui il valore dell'attributo non sia costante,

ma provenga dal campo F_m dalla base di dati, aggiungiamo all'insieme $\mathbf{S}_n$ la condizione $T_{h..ki}.F_m = table\_t_{h..ki}.f_m$;

- $\mathbf{S}^1 = \mathbf{S} \cup \mathbf{S}_n$;
- Per ogni attributo composto $A_{h..kin}$, definito nell'attributo composto $A_{h..ki}$, eseguiamo la funzione InstanceTable ($A_{h..kin}$, $\mathbf{S}^1$, $T_{h..ki}$).

Nel prototipo iniziale l'algoritmo definito per la generazione di pagine statiche non si preoccupava delle varie viste definite dall'utente, in quanto su di esse non veniva fatta alcun tipo di operazione. Le viste venivano materializzate, cioè eseguite a tempo statico, e utilizzate come tabelle presenti nella base di dati; quindi da queste tabelle venivano estratti i valori degli attributi non costanti. Alla fine tutte le viste materializzate venivano rimosse dalla base di dati.

In particolare per ogni attributo composto di livello *i*, al fine di mantenere il legame con i livelli precedenti, si effettuava l'equi-join tra la tabella dalla quale estrarre tutti i valori degli attributi e le tabelle dominanti dei livelli precedenti.

Nel nuovo prototipo realizzato tutto ciò perde di significato in quanto occorre lavorare direttamente sulle viste definite dall'utente; si fa infatti eseguire (a tempo dinamico) la query relativa alla vista del livello *i* e successivamente, sempre a tempo dinamico, quindi in fase di visualizzazione della pagina (per "visualizzazione" si intende la modalità di creazione della pagina dal server su richiesta del client), vengono determinate le condizioni di join con il livello immediatamente precedente (*i-1*), operando direttamente sui cursori run-time.

Per comprendere meglio il concetto immaginiamo di trovarci di fronte a tre liste nidificate, una di livello 0, una di livello *i* e un'altra di livello (*i+1*). Alla vista corrispondente alla lista di livello *i* vengono aggiunte le condizioni di join con il livello 0 e allo stesso modo alla vista di livello (*i+1*) vengono aggiunte solo le condizioni di join con il livello *i*.

Si realizza così un nuovo algoritmo di generazione, all'interno del quale vengono gestiti in maniera completamente diversa gli attributi complessi. Possiamo riassumere per punti ciò che svolge il nuovo algoritmo :

- opera sulle viste definite dall'utente nella clausola USING di ciascun page-scheme effettuando il parsing della query e generando opportune strutture dati ausiliarie;
- effettua l'integrazione tra la struttura ausiliaria precedentemente generata e quella già esistente ottenuta dall'analisi sintattica e semantica del PL;
- genera la pagina dinamica con all'interno le istruzioni derivate dall'algoritmo per la determinazione delle condizioni di join tra i diversi livelli di nidificazione.

4.5 Algoritmo per la costruzione delle query

Durante la definizione del nuovo prototipo, quando ci siamo trovati di fronte a dover verificare quanto del precedente approccio potesse essere importato, abbiamo deciso che occorreva implementare una soluzione ex-novo, infatti se da un lato il precedente approccio semplificava molto le cose dal punto di vista implementativo, dall'altro materializzando le viste nella base di dati poneva dei seri limiti. Il problema maggiore riscontrato da un'analisi preliminare, nell'usare il vecchio approccio, riguardava il decadimento delle prestazioni nel caso in cui veniva generato un sito dinamico; infatti il dover materializzare dei cursori, che possono essere costituiti da un numero elevato di tuple, a tempo dinamico, quindi in fase di visualizzazione della pagina (JSP o ASP), richiedeva un tempo di computazione eccessivamente alto e come conseguenza forniva basse prestazioni. Il risultato di questa analisi preliminare è stato quindi quello di definire un nuovo algoritmo di gestione delle viste presenti nella clausola USING. Questo ha richiesto la definizione e l'implementazione di un parser SQL attraverso il quale è stato possibile estrapolare, data una query, tutte le informazioni in essa contenute (Vedi Paragrafo 4.7).

Descriviamo di seguito l'algoritmo utilizzato per costruire le query sia per il livello 0 sia per il generico livello di nidificazione *i*. La struttura dati che deve essere inizializzata è la `TableStructure` che è definita per mezzo dei seguenti oggetti:

Oggetti istanziati dal parser SQL:

- *queryFields* vettore dei campi utilizzati con elementi "field [AS fields]";

- *queryusedFields* vettore dei campi utilizzati con elementi "field";
- *queryaliasFields* vettore contenente gli alias dei campi utilizzati con elementi "*aliasField*"

Oggetti Restituiti dall'algoritmo:

- *tableName* nome della vista corrispondente alla *tabella dominante* di questo livello;
- *usedFields* vettore dei campi utilizzati o meglio degli attributi che devono essere estratti;
- *fromTables* vettore delle tabelle coinvolte;
- *joinConditions* vettore delle condizioni di join;
- *groupBy* vettore delle condizioni di ragruppamento;
- *Having* vettore delle condizioni aggregate;
- *orderBy* vettore degli attributi, definiti nella vista, rispetto ai quali effettuare l'ordinamento;

Per gli attributi di tipo composto:

- *vista* nome della vista per essi definita;
- *isTable* booleano che fornisce informazione se è una tabella (false) o una vista (true);
- *attributeList* lista delle strutture `TableStructure` associate per gli attributi nidificati;
- *distinct* flag distinct;
- *order* vettore dei campi rispetto ai quali effettuare l'ordinamento. Questo vettore è definito nella clausola ORDER del page-scheme che si sta considerando;
- *isUnique* booleano che permette di determinare se il page-scheme è unico o meno. Nel caso che non lo sia occorre considerare per il livello 0 anche le condizioni d'istanza della pagina.

L'algoritmo di costruzione della query è il seguente:

1. memorizza nel vettore usedField solo i campi degli attributi semplici presenti in questo livello;
2. cerca l'attributo del livello precedente con il quale affettuare l'equi-join e aggiungi tale condizione al vettore joinConditions;
3. SE la pagina non è unica
4. ALLORA aggiungi al vettore joinCoditions la condizione iniziale per l'istanza della pagina;
5. Costruisci la Query nel seguente modo:
6. SE la stringa distinct non è nulla
7. ALLORA aggiungi al vettore SELECT tale stringa
8. FINCHE' il vettore usedField ha un elemento x
9. Aggiungi l'elemento x al vettore *SELECT*
10. FINCHE' il vettore fromTables ha un elemento y
11. Aggiungi l'elemento y al vettore *FROM*
12. FINCHE' il vettore joinConditions ha un elemento z
13. Aggiungi l'elemento z al vettore *JOINCONDITIONS*
14. FINCHE' il vettore order ha un elemento w
15. Aggiungi l'elemento w al vettore *ORDER*
16. Costruisci la Query che è uguale a SELECT+FROM+JOINCONDITIONS+ORDER

Dove SELECT = "SELECT distinct *usedFields*"
FROM = "FROM *fromTables*"
WHERE = "WHERE *joinConditions*"
ORDERBY = "ORDER BY *order*"

In definitiva quello che si ottiene è una stringa, eventualmente parametrica, che deve essere inserita nel programma Java o nelle pagine JSP/ASP generate automaticamente. Occorre osservare che la determinazione delle condizioni di join di cui si parla al punto 2 vengono generate a tempo statico attraverso un secondo algoritmo che viene illustrato di seguito.

4.6 Algoritmo per la gestione degli attributi composti

Illustriamo come vengono determinate le condizioni di Join tra attributi composti nidificati per la generazione di programmi. L'algoritmo viene suddiviso in due passi, un passo base che agisce sugli attributi di tipo composto definiti nel liv.0 e un passo iterativo che agisce sugli attributi composti di livello *i* (con i>=2).

Premettiamo che un attributo composto di livello 0 viene considerato un attributo di liv. 1.

Passo base (i = 1):

Per ogni attributo composto che viene individuato nel Liv.0, si devono eseguire le seguenti azioni:

1. Applica l'algoritmo per la costruzione delle query

2. Scrivi su file[4] la query del Liv.(i-1): Query_(i-1)
3. Scrivi su file come eseguire la query Query_(i-1)
4. Scrivi come aprire il cursore_(i-1)
5. Per ogni attributo verifica:
6. SE (Tipo_Attributo != "LIST-OF")
7. ALLORA Scrivi su file le istruzioni per stampare il valore dell'attributo in esame
8. ALTRIMENTI Gestione_LISTOF(Query_(i-1), NomeLista)

La funzione Gestione_LISTOF prende in input, il nome dell'attributo composto e il nome della lista.

Passo iterativo (i >= 2):

Per ogni attributo di tipo composto nidificato rispetto al Liv.0 si invoca la seguente funzione ricorsiva Function Gestione_LISTOF(Query_(i-1), NomeLista):

1. Determina gli attributi ritornati da Query_i inizializzando il vettore attributi_i
2. Determina gli attributi ritornati da Query_(i-1) inizializzando il vettore attributi_(i-1)

[4] per file si intende il sorgente generato in codice Java, JSP o ASP

3. FINCHE' il vettore attributi_(i-1) non è terminato:
4. Per ogni attributo A_c ritornato da Query_i (Per la scansione si usa l'indice c)
5. Per ogni attributo A_j ritornato da Query_(i+1) (Per la scansione si usa l'indice j)
6. SE Il nome dell'attributo in posizione c-esima coincide con quello in posizione j-esima (A_c.Name = A_j.Name)
7. ALLORA aggiungi al vettore delle condizioni A_j.Name = "valore assunto dall'attributo A_c "
8. Applica l'algoritmo per la costruzione delle query
9. Scrivi su file la query del Liv.i di nome Query_i
10. Scrivi su file le operazioni per eseguire la query Query_i
11. Scrivi su file come aprire il cursore_i
12. Per ogni attributo del Liv.i verifica
13. SE (Tipo_Attributo != "LIST-OF")
14. ALLORA Scrivi su file le istruzioni per stampare il valore dell'attributo in esame
15. ALTRIMENTI Gestione_LISTOF(Query_i, NomeLista)

4.7 Definizione ed implementazione di un parser SQL

Come descritto nel paragrafo 4.5 al fine di fornire un nuovo algoritmo di generazione è stato necessario definire ed implementare un parser SQL che permette di operare sia sugli attributi ritornati dalla query, sia sulle tabelle dalle quali si estraggono tali attributi e sia sulle eventuali condizioni definite nella clausola Where. La classe Java adibita al parser delle query è la `ParserQuery`. Illustriamo di seguito la grammatica definita per il parser SQL:

View → "SELECT" selectClause
"FROM" fromClause
["WHERE" whereClause]
["GROUP BY" groupbyClause]
["HAVING" havingClause]
["ORDER BY" orderbyClause]

selectClause	→	<field> [“AS” <field>] (, <field> [AS <field>])*
fromClause	→	<tables> [AS <tables>] (, <tables> [AS <tables>])*
whereClause	→	constValue (,constValue)*
groupbyClause	→	constValue (,constValue)*
havingClause	→	constValue (,constValue)*
orderbyClause	→	constValue (,constValue)*
constValue	→	"" value ""

Come si può notare dalla grammatica le condizioni specificate nella whereClause, la lista degli attributi di raggruppamento nella groupbyClause, la lista delle condizioni aggregate nella havingClause ed infine la lista degli attributi di ordinamento nella orderbyClause, vengono parsati come valori costanti.

Descritta la grammatica utilizzata, possiamo illustrare come viene utilizzato il parser SQL. Presa in input una query possiamo scomporre il lavoro del parser in due fasi :

1. La query viene decomposta nelle sue parti fondamentali attraverso i token SELECT, FROM e se presenti WHERE, GROUP BY, HAVING, ORDER BY.
2. Una volta decomposta la query nelle sue parti principali esse vengono parsate al fine di estrarre le informazioni di interesse.

Ad esempio supponiamo di parsare la clausola SELECT e supponiamo di avere:

```
SELECT VOLUME, ARTICLES.NUM, MIN(POSITION) AS SECTIONPOSITION
```

Su ogni singolo attributo della clausola SELECT (ed anche della FROM) viene determinato:

1. il nome del campo (o tabella nel caso della FROM);
2. se presente il suo alias;
3. il nome del campo (o tabella) AS il suo alias (sempre che sia presente).

Quindi nel nostro esempio per il terzo attributo della clausola SELECT estrarremo:

`MIN(POSITION)`, `SECTIONPOSITION` e infine `MIN(POSITION) AS SECTIONPOSITION`.

Questi tre elementi sono item, posti in posizione omonima, di tre vettori costruiti in memoria. Entrando più nel dettaglio, illustriamo di seguito la struttura dati costruita in memoria in memoria come risultato del nostro parser SQL:

- *querytableName* stringa identificativa della vista;
- *queryFields* Vettore dei campi utilizzati riempito con stringhe del tipo “field [AS aliasfield]”;
- *queryusedFields* Vettore dei campi utilizzati riempito con stringhe del tipo “field”;
- queryaliasFields Vettore dei campi utilizzati riempito con stringhei che rappresentano gli alias dei campi utilizzati quindi con gli aliasField;
- *queryTables* Vettore delle tabelle coinvolte riempito con stringhe della forma “table [AS table]”;
- *queryfromTables* Vettore delle tabelle coinvolte riempito con stringhe del tipo “table”;
- *queryaliasTables* Vettore delle tabelle coinvolte riempito con gli alias delle tabelle coinvolte quindi con gli “aliasTable”;
- *queryjoinConditions* Vettore delle condizioni di join;
- *queryOrder* Vettore degli attributi rispetto al quale effettuare l’ordinamento;
- *queryGroup* Vettore degli attributi di raggruppamento;
- *queryHaving* Vettore delle condizioni aggregate;
- *definitionView* stringa che contiene la definizione della vista;

Se ad esempio supponiamo di aver definito la seguente query:

```
SECTIONS:{SELECT   VOLUME, ARTICLES.NUM, ARTICLES.SECTION,
                   MIN(POSITION)AS SECTIONPOSITION
          FROM     ARTICLES
          GROUP BY VOLUME, NUM, SECTION} DISTINCT
```

Le fasi di lavoro del parser sono schematizzate nella figura 4.9. Come si vede nella prima fase vengono estratte le varie clausole dando luogo alle stringhe SELECT, FROM, WHERE, GROUPBY, HAVING, ORDERBY che contengono le omonime clausole. Nella seconda fase le stringhe ottenute vengono parsate in modo tale da ottenere dei vettori i cui elementi sono gli attributi in esse presenti con gli eventuali alias.

Come si vede dall'esempio nel caso in cui l'attributo non presenta ridenominazione nei tre vettori viene sempre riportato il suo nome al fine di mantenere consistenza nelle strutture dati.

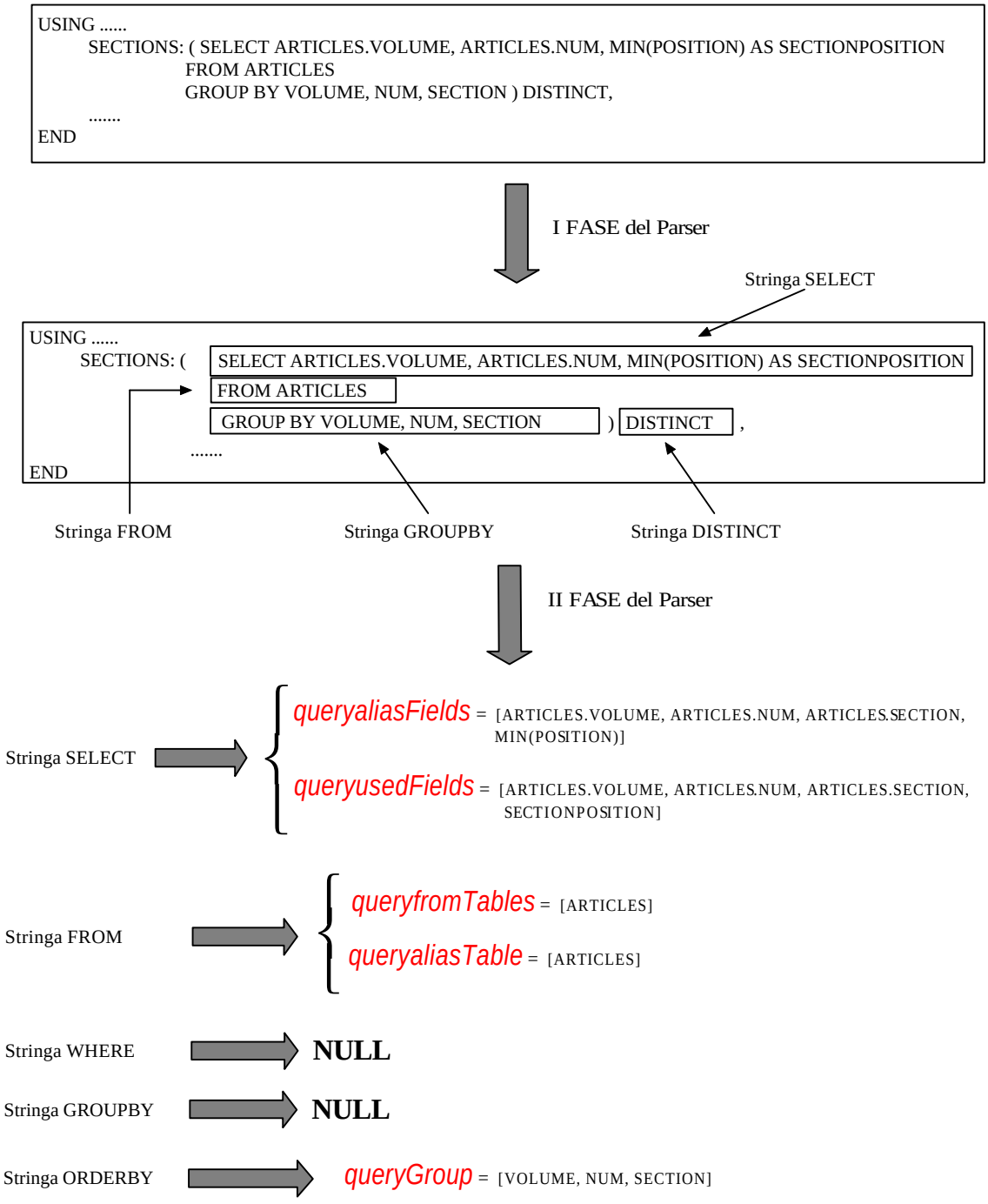

Figura 4.9: Fasi del parser SQL

4.8 Integrazione tra le strutture dati Penelope

Nei precedenti paragrafi sono state illustrate le strutture dati che vengono istanziate dai moduli adibiti al parsing e al controllo semantico del PL e quelle generate dal parser SQL, al fine di ottenere una serie di strutture utili al nostro scopo è stata effettuata attraverso la classe Java MappingStructure, una loro integrazione.
Riferiamoci al seguente page-scheme:

```
DEFINE PAGE EducationPage : "Education"
AS URL("indexEducation");
   ...
USING CorsoLaurea,
      Prof : (SELECT Cl, Professore.Cognome, Nome, email
              FROM Professore),
      ProfCourse : (SELECT  Cognome, CorsoNome
                    FROM Professore, Corsi
                    WHERE Professore.Cognome=Corsi.Docente)
ORDER BY CorsoNome

END
```

Viste definite

Di seguito sono riportate le informazioni delle viste, definite nella clausola USING, così come vengono memorizzate nella struttura TableStructure.

Sulla sinistra viene riportato l'oggetto TableStructure istanziato dal parser Penelope, mentre sulla destra viene riportato il medesimo oggetto modificato dalla MappingStructure.

OGGETTO TABLESTRUCTURE INIZIALE

vista	
isTable	= false
tableName	= CL
usedFields	= [CL.Cl]
fromTables	= [CL]
joinConditions	[]
groupBy	[]
Having	[]
orderBy	[]
distinct	

MappingStructure

OGGETTO TABLESTRUCTURE MODIFICATO

vista	
isTable	= false
tableName	= CL
usedFields	= [CL.Cl]
fromTables	= [CL]
joinConditions	[]
groupBy	[]
Having	[]
orderBy	[]
distinct	

Figura 4.10: Oggetto TableStructure per la prima Vista definita nella clausola USING

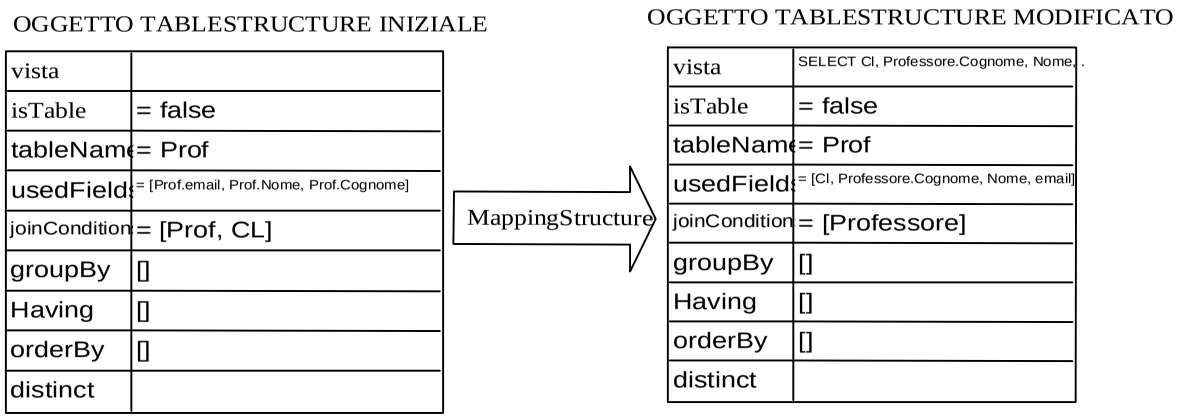

Figura 4.11: Oggetto TableStructure per la seconda Vista definita nella clausola USING

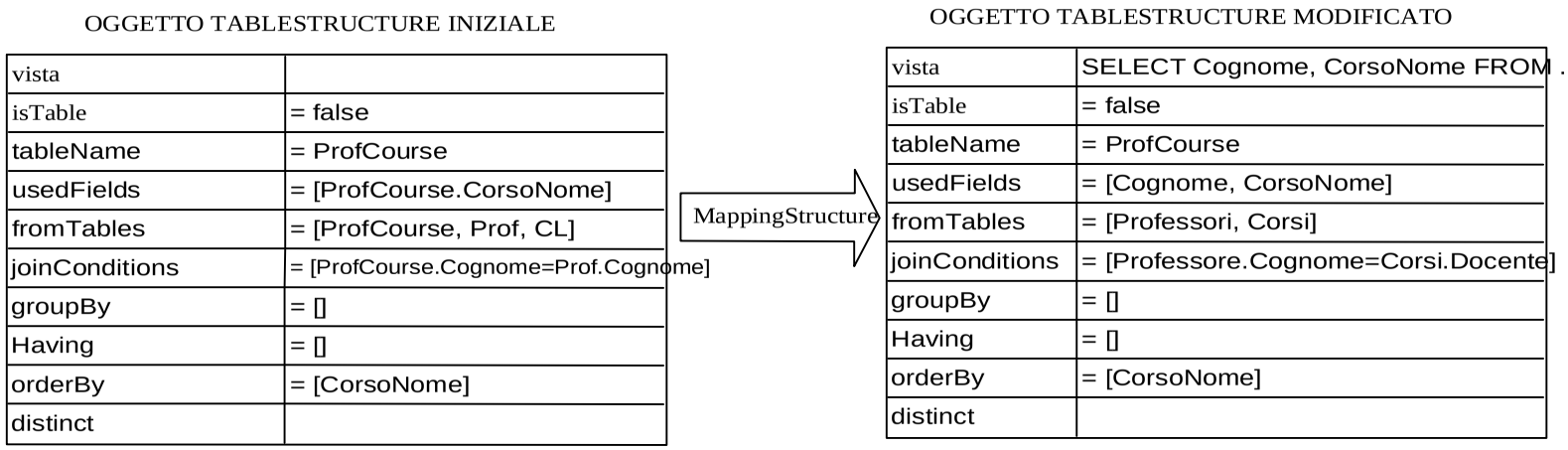

Figura 4.12: Oggetto TableStructure per la terza Vista definita nella clausola USING