Capitolo 3

Il nuovo prototipo Penelope

Nel corso di questo capitolo verranno fornite le linee guida attraverso le quali è stato possibile definire ed implementare nuovi algoritmi e nuove strutture dati al fine di proporre una nuova versione del prototipo Penelope.

Verrà illustrata l'architettura globale del sistema Penelope (in modo da averne una visione globale), e verranno evidenziate le caratteristiche dei moduli di cui è composta per poi passare, nel capitolo successivo, a descrivere le strutture dati che i singoli moduli dell'architettura generano in memoria e i vari algoritmi di generazione che sono stati definiti e implementati.

3.1 Architettura del sistema Penelope

L'architettura del sistema può essere schematizzata in maniera modulare come mostrato nella figura 3.1.

Il modulo Analizzatore Sintattico riceve in input il file PL (file all'interno del quale sono definite le istruzioni Penelope) e verifica la correttezza delle istruzioni in esso contenute; l'Analizzatore Semantico verifica la fattibilità di esecuzione delle stesse e quindi fornisce al modulo Generatore le strutture dati necessarie per la produzione di programmi.

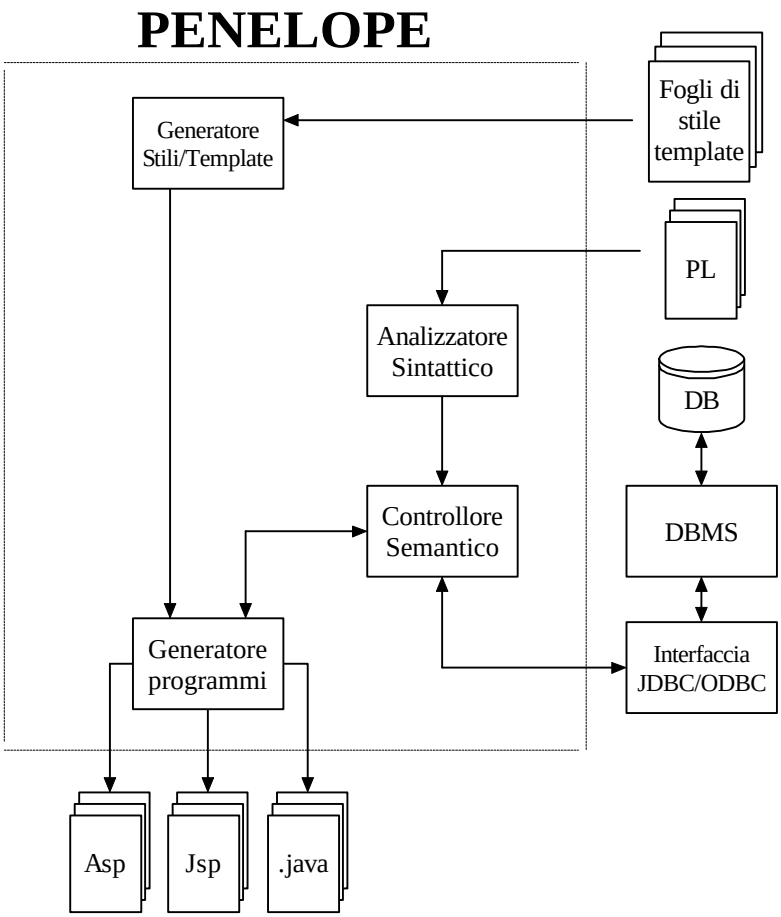


Figura 3.1: Architettura del prototipo Penelope

Di seguito riportiamo una descrizione sintetica del ruolo svolto dai moduli più importanti.

3.1.1 Analizzatore Sintattico

L'Analizzatore Sintattico è il modulo del sistema che verifica la correttezza sintattica delle istruzioni fornite in input attraverso il PL.

In particolare, l'analizzatore apre il file PL contenente le istruzioni, verifica la correttezza delle strutture dati/ipertestuali in esso contenute e ne genera una rappresentazione in memoria. Alla fine il controllo viene trasferito al Controllore Semantico.

Riportiamo di seguito i principali metodi che vengono utilizzati da tale modulo evidenziando l'input e l'output:

- Il metodo **parse** ha il compito di eseguire il parser sintattico del file con il quale il modulo è stato inizializzato. Il suo output è un eccezione nel caso in cui il file di input contenga istruzioni sintatticamente scorrette; altrimenti un vettore contenente le strutture dati relative alle istruzioni contenute nel file di input.
- Il metodo **close** rilascia le risorse utilizzate dall'analizzatore sintattico.

3.1.2 Controllore Semantico

Il Controllore Semantico verifica la correttezza delle relazioni che sono definite tra lo schema logico della base di dati e l'equivalente modello logico navigazionale.

Ciò è necessario per stabilire se effettivamente nella base di dati siano contenute e rese disponibili tutte le informazioni che si intendono pubblicare in formato multimediale.

Il modulo effettua quindi una serie di controlli preliminari sullo schema della base di dati e sulle strutture in memoria prima di passare queste ultime al modulo Generatore.

Anche per questo modulo riportiamo di seguito i principali metodi che vengono utilizzati:

- Il metodo ***init*** si occupa di inizializzare il controllore semantico. Prende in input due parametri:
    1. *structure*: rappresenta le strutture restituite dall'analizzatore sintattico contenenti informazioni sulle istruzioni del file analizzato.
    2. *Db*: rappresenta l'interfaccia JDBC/ODBC relativa alla base di dati contenente i dati che si intendono pubblicare.

- Il metodo ***check*** che esegue il controllo semantico delle strutture dati, controlla la coerenza delle stesse mediante un confronto con lo schema della base di dati, ed infine aggiorna le strutture sulla base delle verifiche effettuate. L'output restituito

è un eccezione nel caso in cui le strutture dati corrispondano ad istruzioni non valide semanticamente.

- Il metodo **close** rilascia le risorse utilizzate dal controllore semantico.

3.1.3 Interfaccia JDBC/ODBC

L'interazione JDBC/ODBC realizza il disaccoppiamento con la base di dati. Qualsiasi modulo del sistema che abbia necessità di interagire con la base di dati invoca solo ed unicamente i metodi messi a disposizione da questa interfaccia.

3.1.4 Generatore Stili/Template

Il modulo ha il compito di generare una presentazione prototipale per ciascun page-scheme definito nello schema logico del sito. La presentazione può essere modificata attraverso il package Telemaco al fine di personalizzarla e migliorarla.

Il modulo inoltre ha il compito di relazionare la presentazione grafica alla struttura logica della pagina per realizzare la pubblicazione dei dati.

3.1.5 Generatore di programmi

Il modulo Generatore ha il compito di creare fisicamente i programmi a partire dal file PL. Si ricorda che per programmi, in questo contesto, si intendono file JSP, ASP oppure Java.

Al fine di rendere Penelope il più possibile modulare, il Generatore di programmi è stato suddiviso in due ulteriori moduli:

1. ***GenerateASP*** che si occupa di generare pagine ASP;
2. ***GenerateSources*** che si occupa di generare sia programmi Java che file JSP.

Entrambi i moduli utilizzano le strutture dati che sono create in memoria dall'analizzatore sintattico e da quello semantico.

3.2 Architettura della classe PenelopePDL

Il prototipo Penelope ha una struttura fortemente modulare, al suo interno possono essere individuati diversi moduli fondamentali. Tra tutti, vi è la classe PenelopePDL che svolge il ruolo di modulo guida.

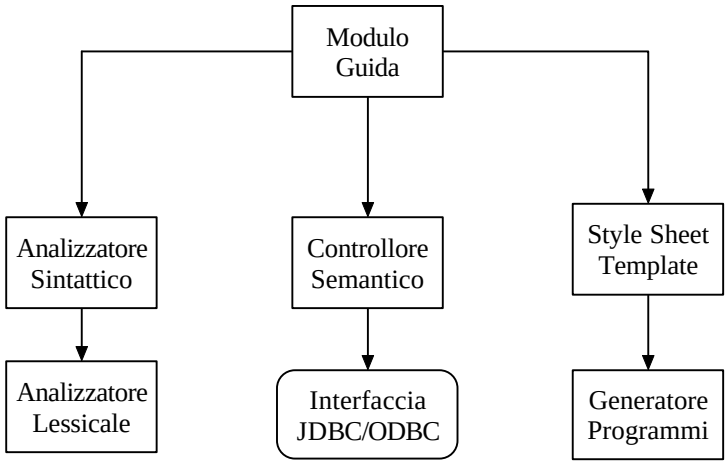


Figura 3.2: Moduli del Package Penelope

E' proprio da questa classe che vengono invocati i moduli preposti al controllo semantico e sintattico del PL. Naturalmente se i controlli hanno esito favorevole viene invocato il modulo di generazione dei programmi altrimenti vengono sollevate le eccezioni corrispondenti al tipo di errore che si è verificato.

3.3 Funzionamento di Penelope

Descritta l'architettura di Penelope si vuole ora mostrare come opera l'interprete Penelope. All'interno di un page-scheme distinguiamo due tipi di attributi:

1. attributi di livello zero(0) ;
2. attributi di livello "*i*".

Gli attributi del secondo tipo vengono individuati quando all'interno del page-scheme sono definiti degli attributi composti LIST-OF, FORM, MAP. Di seguito viene riportato un page scheme all'interno del quale si sono individuati gli attributi

a diversi livelli di nidificazione:

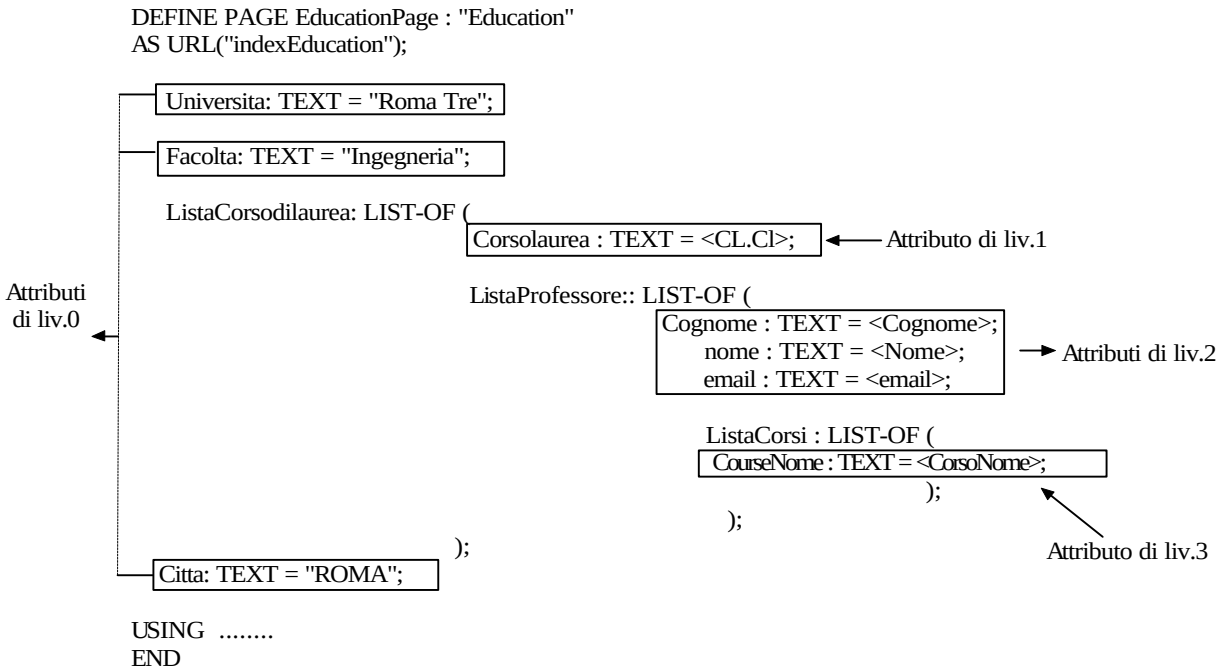


Figura 3.3: Individuazione degli attributi Penelope a diversi livelli

Consideriamo ora un generico page-scheme definito in un PL; nella clausola USING sono presenti tante viste quanti sono gli attributi composti definiti all'interno del PL. Ogni vista definita per un certo livello di nidificazione *i* deve mantenere una relazione con il precedente livello *i*, basti pensare a due liste nidificate.

Questo requisito è soddisfatto imponendo che nella vista più interna sia estratto l'attributo con il quale è possibile fare l'equi-join con la vista del livello immediatamente precedente.

Cerchiamo di far capire meglio quanto detto attraverso un semplice esempio.

ESEMPIO

Supponiamo di avere a disposizione una base di dati nella quale siano memorizzate le seguenti tabelle relazionali:

Cl
Informatica
Elettronica
Meccanica

Figura 3.4: Tabella relazionale CorsoLaurea

Cognome	Nome	email	Cl
Atzeni	Paolo	atzeni@dia..	Informatica
Di Battista	Giuseppe	gdb@dia...	Informatica
Neri	Alessandro	neri@ele...	Elettronica
Assanto	Gaetano	assanto@ele.	Elettronica
Cerri	Giovanni	cerri@...	Meccanica
Cialdea	Marta	cialdea@dia.	Informatica

Figura 3.5: Tabella relazionale Professore

Cognome	CorsoNome
Atzeni	Basi di Dati
Atzeni	Sistemi Informativi
Di Battista	Informatica Terorica
Di Battista	Impianti di Elaborazione
Neri	Teoria dei Segnali
Cialdea	Fondamenti II
Assanto	Optoelettronica
Cerri	Meccanica Appl..
Cialdea	Intelligenza Artificiale

Figura 3.6: Tabella relazionale Corsi

Supponiamo inoltre di aver definito il seguente page-scheme che consente di ottenere per ogni corso di laurea la lista dei professori che ad esso fanno riferimento con i relativi corsi che tengono.

```
DEFINE PAGE EducationPage : "Education"
AS URL("indexEducation");
   Universita: TEXT = "Roma Tre";
   ListaCorsodilaurea: LIST-OF (
                                 Corsolaurea : TEXT = <CL.Cl>;
                                 ListaProfessore:: LIST-OF (  Cognome : TEXT = <Cognome>;
                                                              nome : TEXT = <Nome>;
                                                              email : TEXT = <email>;
                                                              ListaCorsi : LIST-OF (
                                                               CourseNome : TEXT = <CorsoNome>;
                                                                                   );
                                                                );
                               );
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

Nell'esempio l'attributo Cl, che viene estratto nella seconda vista definita, è l'attributo con il quale effettuare il join con la tabella del livello precedente "CorsoLaurea". Allo stesso modo nella vista ProfCourse viene estratto l'attributo Cognome che permette di fare il join con la vista Prof.

Tali condizioni vengono determinate dinamicamente in fase di visualizzazione della pagina, cioè in fase dinamica verranno eseguite le query definite nella clausola USING opportunamente rese parametriche rispetto al livello precedente:

CorsoLaurea	Elaborazione Penelope →	SELECT Cl FROM CorsoLaurea
Prof	Elaborazione Penelope →	SELECT Cl, Professore.Cognome, Nome, email FROM Professore WHERE Cl = <Valori assunti da "Cl" nel cursore CorsoLaurea>
ProfCourse	Elaborazione Penelope →	SELECT Cognome, CorsoNome FROM Professore, Corsi WHERE Professore.Cognome = Corsi.Docente AND Professore.Cognome = <Valori assunti da "Cognome" nel cursore Prof>

Figura 3.7: Query elaborate da Penelope

Per ogni cursore ottenuto viene eseguita una query parametrica rispetto al livello successivo. Il cursore che si ottiene per la “CorsoLaurea” coincide con la tabella relazionale stessa riportata in figura 3.4, questo si traduce, per la seconda LIST-OF, nell’eseguire tre volte la query relativa a “Prof”:

```
SELECT Cl, Professore.Cognome, Nome, email
FROM Professore
WHERE Cl = <Valori assunti da "Cl" nel cursore CorsoLaurea>
```

dove Cl assume valori “Informatica”, “Elettronica”, “Meccanica”. I tre cursori che si ottengono sono:

Per Cl = 'Informatica'

Cognome	Nome	email	Cl
Atzeni	Paolo	atzeni@dia..	Informatica
Di Battista	Giuseppe	gdb@dia...	Informatica
Cialdea	Marta	cialdea@dia.	Informatica

Per Cl = 'Elettronica'

Cognome	Nome	email	Cl
Neri	Alessandro	neri@ele...	Elettronica
Assanto	Gaetano	assanto@ele.	Elettronica

Per Cl = 'Meccanica'

Cognome	Nome	email	Cl
Cerri	Giovanni	cerri@...	Meccanica

Figura 3.8: Cursori ottenuti per la vista “Prof”

Allo stesso modo per la “ProfCourse”, terza LIST-OF, viene eseguita sei volte la query relativa:

```
SELECT Cognome, CorsoNome
FROM Professore, Corsi
WHERE Professore.Cognome = Corsi.Docente AND
        Professore.Cognome = <Valori assunti da "Cognome" nel cursore Prof>
```

dove *Professore.Cognome* assume valori “Atzeni”, “Di Battista”, “Cialdea”, “Assanto”, “Neri”, “Cerri”. I cursori che si ottengono sono:

Cognome	CorsoNome
Atzeni	Basi di Dati
Atzeni	Sistemi Informativi

Cognome	CorsoNome
Di Battista	Informatica Teorica
Di Battista	Impianti di Elaborazione

Cognome	CorsoNome
Cialdea	Intelligenza Artificiale

Cognome	CorsoNome
Assanto	Optoelettronica

Cognome	CorsoNome
Neri	Teoria dei Segnali

Cognome	CorsoNome
Cerri	Mecccanica Appl..

Figura 3.9: Cursori ottenuti per la vista “ProfCourse”

La pagina generata fornisce il seguente output:

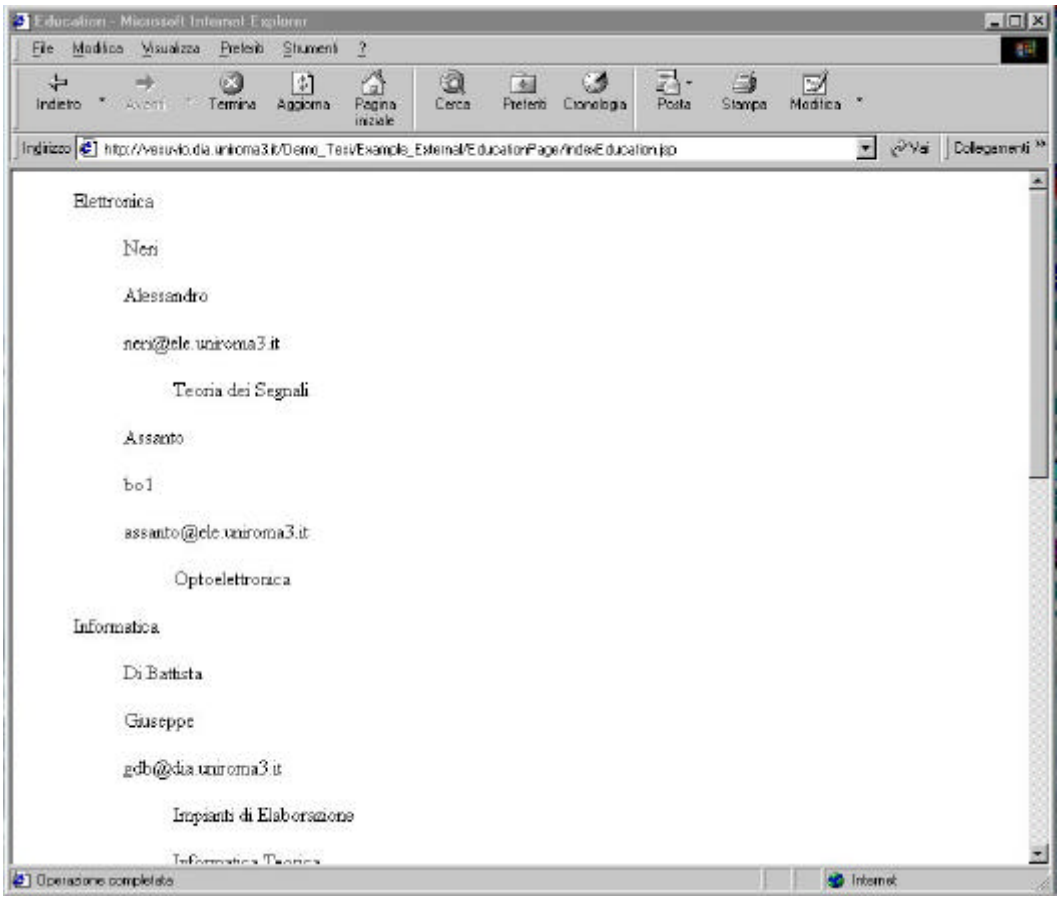


Figura 3.10: Pagina generata da Penelope e visualizzata con un browser

3.4 Idee base per la generazione automatica di pagine dinamiche

Per semplicità indichiamo con il termine SP (Server Page) la generica pagina dinamica che può quindi essere nel nostro caso una JSP oppure una ASP. Illustriamo di seguito quali sono state le linee guida che si sono seguite per generare pagine dinamiche.

- Per ogni page-scheme deve essere creata una SP;
- Se il page-scheme non è costante occorre aprire una connessione con la base di dati;
- Per ogni page-scheme, non costante, va creata almeno una query SQL che ci consenta di ottenere tutti e soli gli attributi necessari a riempire le pagine;
- Servono tante query quanti sono gli attributi composti presenti nel page-scheme;
- Se il page-scheme non è unico, le query devono essere parametriche;

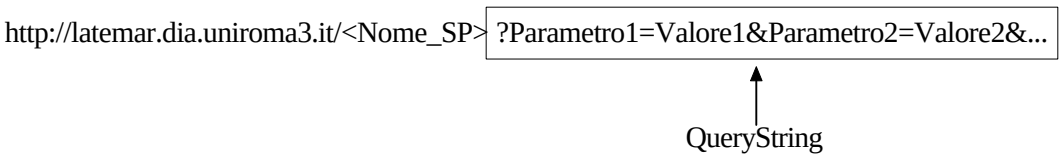
- I link ADM corrispondono a link HTML/XML parametrici tra le SP coinvolte;

Vediamo come tutto questo può essere tradotto nel caso di page-scheme non costante e nel quale siano presenti solo attributi semplici :

```
<HTML><HEAD>…</HEAD></BODY>
ApriConnessione
Imposta ed esegui Query
Cattura la Tabella Risultato della Query
Controlla che la tabella risultato non sia vuota
Apri Tabella Risultato
Vai alla Prima Riga
<UL>
Finchè non incontri la fine del risultato
    <LI> stampa valore Attributo 1 <BR>
         stampa valore Attributo 2 <BR>
         …
         stampa valore Attributo n <BR>
    Vai alla Riga Successiva
Loop
</UL></BODY></HTML>
```

Per prima cosa si deve aprire la connessione con la base di dati, dopo di che si definisce e si esegue la query per estrarre gli attributi semplici di livello 0. Viene catturata la tabella risultato e ci si posiziona sulla prima riga se e solo se il cursore è non vuoto. Si cicla finché non è terminato il cursore.

Nel caso in cui si debba generare un page-scheme non unico, allora la pagina generata verrà invocata ad esempio attraverso il seguente URL:



Bisognerà quindi prevedere anche la cattura dei parametri (QueryString) attraverso la quale verrà invocata la pagina. Questi parametri verranno utilizzati per costruire la query di livello 0.

```
<HTML><HEAD>…</HEAD></BODY>
Cattura la QueryString
ApriConnessione
Imposta ed esegui Query effettuando le selezioni sulla
                                    base dei parametri passati
Cattura la Tabella Risultato della Query
.....
</UL></BODY></HTML>
```

Traduzione degli attributi LINK-TO di Penelope in SP:

```
<HTML><HEAD>…</HEAD></BODY>
ApriConnessione
Imposta Query
Cattura la Tabella Risultato della Query
Controlla che la tabella risultato non sia vuota
Apri Tabella Risultato
Vai alla Prima Riga
<UL>
Finchè non incontri la fine del risultato
    <LI><A HREF ="nextpage.sp?Parametro=
        stampa valore Parametro">
        stampa valore Attributo 1 <BR>
        stampa valore Attributo 2 <BR>
        …
        stampa valore Attributo n <BR>
    Vai alla Riga Successiva
Loop
</UL></BODY></HTML>
```

Traduzione di un page-scheme non unico in SP:

```
<HTML><HEAD>…</HEAD></BODY>
Prendi il parametro
ApriConnessione
Imposta Query con il parametro
Cattura la Tabella Risultato della Query
Controlla che la tabella risultato non sia vuota
Apri Tabella Risultato
Vai alla Prima Riga
<UL>
Finchè non incontri la fine del risultato
    <LI> stampa valore Attributo 1 <BR>
         stampa valore Attributo 2 <BR>
         …
         stampa valore Attributo n <BR>
    Vai alla Riga Successiva
Loop
</UL></BODY></HTML>
```

3.5 Generazione automatica di pagine ASP

Le ASP sono la risposta della Microsoft alla esigenza di dotare i web server di una capacità di programmazione per la gestione di pagine dinamiche. Ovviamente nell'intento di offrire un cammino il più possibile indolore verso i propri sistemi, Microsoft non ha potuto trascurare il fatto che in ambiente internet erano già consolidati dei sistemi di programmazione basati su JavaScript e che comunque uno dei linguaggi più usati era il linguaggio Perl.

Per questo motivo sin dall'inizio le asp sono state progettate per essere multi linguaggio: così come sono fornite, sono in grado di utilizzare direttamente sia il VBScript che la versione Microsoft del JavaScript, ovvero Jscript: il programmatore deve solamente specificare, con una direttiva <%@ language=Jscript"> quale linguaggio usare per lo sviluppo delle pagine.

Il diffondersi sempre più incessante di siti asp a fatto si che il nuovo prototipo Penelope prevedesse la possibilità di generare in maniera automatica siti ASP.

Entriamo in dettaglio circa le scelte utilizzate per generare pagine ASP.

Cominciamo con il dire che il linguaggio scelto per sviluppare tali pagine è il VBScript. Si è optato per questa scelta essenzialmente per due motivi:

1) quasi tutti i siti ASP esistenti utilizzano all'interno delle loro pagine il VBScript;
2) VBScript, insieme al Visual Basic, costituisce il linguaggio di punta della Microsoft.

In questo paragrafo verrà descritto come si è operato al fine di generare automaticamente attraverso Penelope pagine ASP, senza entrare nei dettagli della sintassi ASP (per eventuali chiarimenti ed approfondimenti è possibile far riferimento alla specifica appendice B di questa tesi).

La prima istruzione di scripting che viene inserita all'interno del file generato è <%Response.Expires = 0%> che serve per forzare a non tenere traccia della pagina caricata nella cache; pertanto la pagina viene richiesta dal client, viene visualizzata, ma non viene salvata nella cache.

Dopo di che viene verificato se il page-scheme è unico o meno; se non lo è viene inserito nel file un controllo sulla QueryString passata nell'URL.

Immaginiamo ad esempio di aver generato la pagina IndexTermsPage corrispondente alla seguente definizione iniziale di PageScheme:

```
DEFINE PAGE Dynamic IndexTermsPage : "Index Terms and Cathegories"
    STYLE g:\users\ditomass\wbms43\sites\styleSigmod\SigmodStyle.STY()
  AS URL (<ArtsView.ARTICLECODE>);
  …….
```

Poiché la funzione URL ha un attributo non costante vuol dire che la pagina non è unica e che pertanto quando verrà richiamata l'ASP generata dovrà necessariamente prendere in input un parametro d'istanza ARTICLECODE.

La chiamata a tale pagina avverrà mediante il seguente URL :

http://latemar.dia.uniroma3.it/Sigmod/IndexTermsPage/IndexTermsPage.jsp?ARTICLECODE=00585

↑
QueryString

Il file generato conterrà un Header che è il seguente:

```
<%Response.Expires=0%>
<% If 0= Len(Request.ServerVariables("QUERY_STRING")) THEN %>
 <html><title>Page not available</title> </head><body><CENTER><H2>
                    Wrong parameters</H2></CENTER></body></html>
<% ELSE %>
……
```

Cioè viene subito controllato che la QueryString non abbia lunghezza nulla.

Quanto detto viene fatto attraverso l'istruzione Request.ServerVariables("QUERY_STRING") che permette di prelevare la stringa " ARTICLECODE=00585", e attraverso la funzione Len, che applicata ad una stringa restituisce la sua lunghezza. Nel caso in cui la QueryString presenti lunghezza nulla si visualizza un messaggio di errore che segnala che la pagina richiesta non è disponibile in quanto si sono passati dei parametri errati. Viceversa se non è nulla viene catturata attraverso l'oggetto Request:

```
<% ArtsViewARTICLECODE = Request.QueryString("ARTICLECODE") %>
```

Una volta determinati i parametri sulla base dei quali generare la pagina si verifica se il page-scheme è costante o meno. Ciò significa che se il page-scheme è variabile deve necessariamente essere aperta una connessione con la base di dati in quanto vuol dire che alcuni attributi non sono costanti. La stringa che viene scritta su file è la seguente:

```
<% Set conn = Server.CreateObject("ADODB.Connection")
    conn.open "nome_sorgente_dati"  %>
```

Questa non fa altro che creare un nuovo oggetto ADODB che permette di effettuare la connessione con una sorgente dati ODBC il cui nome è "nome_sorgente_dati".

Poi viene scritta su file la query (nel nostro esempio parametrica) e come eseguirla. Occorre anticipare che le query vengono generate applicando un

determinato algoritmo di generazione che verrà illustrato nel capitolo successivo.

```
<% strQuery = "Select …. From …. Where ….  AND ARTICLES.ARTICLECODE =
'"&cstr(ArtsViewARTICLECODE)&"'"
Set rst = Server.CreateObject("ADODB.Recordset") %>
<% rst.Open strQuery, conn
```

Anche in questo caso è stato inserito un controllo, nell'eventualità che i parametri rispetto ai quali si vuole istanziare la pagina siano errati, in particolare si effettua un controllo sul cursore ritornato, se è nullo allora vuol dire che la query non ha fornito tuple pertanto viene visualizzato un messaggio di errore che segnala che la pagina richiesta non è disponibile in quanto si sono passati dei parametri errati.

```
If rst.eof THEN %>
<html><head><title>Page not available</title></head><body><CENTER>
            <H2>Wrong parameters</H2></CENTER></body></html>
<% ELSE %>
```

Per stampare un attributo, ottenuto come risultato dell'esecuzione di una query, si utilizza l'oggetto *Response* al quale si applica il metodo *write*. Per catturare il valore si utilizza il metodo *Fields("<Nome_Attributo>").Value* che applicato al cursore restituisce il valore assunto dall'attributo.

```
<% Response.Write(rst.Fields("TITLE").Value)%>
```

Immaginiamo ora che nel PL sia presente un attributo composto, ad esempio:

```
DEFINE PAGE Dynamic IndexTermsPage : "Index Terms and Cathegories"
   STYLE g:\users\ditomass\wbms43\sites\styleSigmod\SigmodStyle.STY()
 AS URL (<ArtsView.ARTICLECODE>);
   ……..
```

```
authors     : LIST-OF (
                        author : TEXT = <AUTHORS.AUTHOR>;
                        );
  …….
END
```

Questo sul file ASP generato equivale a scrivere le seguenti istruzioni:

```
1.  <% authorsQuery = "Select … From …. Where
2.       AUTHORS.ARTICLECODE = '"&rst.Fields("ARTICLECODE").Value&"'
3.       Order By AUTHORS.AUTHORPOSITION"
4.       Set authors= conn.execute(authorsQuery) %>
5. <% If (NOT authors.eof) THEN  %>
6. <%    authors.MoveFirst
7.         do while Not authors.eof  %>
8. <%      Response.Write(authors.Fields("AUTHOR").Value) %>
9. <%      authors.MoveNext
10.        loop
11.    END IF
12.    authors.close
13.    set authors = NOTHING   %>
```

Viene definita la query (ottenuta applicando l'algoritmo di generazione delle query descritto nel successivo capitolo) (punto 1, 2, 3), nel punto 2 è evidenziato il fatto che si effettua il join naturale con il livello precedente che può essere il livello zero oppure un precedente livello di nidificazione.

Si esegue la query (punto 4), si controlla che il cursore non sia vuoto con "NOT authors.eof" (punto 5), si posiziona il puntatore sulla prima tupla del cursore con "authors.MoveFirst" (punto 6), si inizia a ciclare finché sono presenti tuple (punto 7), si stampa il valore dell'attributo AUTHOR (punto 8), ci si sposta sulla

successiva tupla (punto 9) e si continua a ciclare (punto 10). Alla fine quando si è scandito l'intero cursore, viene chiuso (punto 12) e rilasciato (punto 13).

Vediamo ora come vengono tradotti gli attributo LINK-TO. Supponiamo di aver definito:

```
Toindex  : LINK-TO IndexTermsPage (
                                    URL(<ARTICLECODE>);
                                    index : TEXT = "…";
                                  );
```

Penelope codifica questo come:

```
<A HREF="../IndexTermsPage/IndexTermsPage.asp?ARTICLECODE=
    <% Response.Write(Server.UrlEncode(articles.Fields("ARTICLECODE").Value))%>  ">
```

Il metodo *Server.UrlEncode* viene utilizzato al fine di codificare gli spazi vuoti (che possono essere presenti) con il "+"; in questo modo il link viene accettato da tutti i browser.

3.6 Generazione automatica di pagine JSP

Java offre ben due soluzioni per la programmazione web: innanzitutto le servlet, che sono l'equivalente Java della programmazione Cgi. Java non può essere utilizzato in maniera efficiente come linguaggio per la programmazione cgi e per ovviare a questo limite è stata sviluppata una specifica, detta Sevlet API, che indica ai web server come fare per utilizzare Java per generare pagine dinamiche.

Le servlet non sono comunque la soluzione a tutto, dato che presentano le stesse difficoltà di uso delle cgi. Per questo motivo sono stati introdotti dei sistemi che semplificano lo sviluppo di applicazioni web simili alle asp: le ***J***ava ***S***erver ***P***ages. Le JSP adottano una sintassi molto simile alle ASP; lo sviluppatore web deve solamente editare una pagina HTML/XML e incorporarvi del codice Java anziché codice di

Script. Tecnicamente c'è una grossa differenze tra le ASP e le JSP. Infatti mentre per le prime viene interpretato direttamente il codice, JSP compila *on-the-fly* le pagine in una servlet.

Esistono due possibilità per accedere ad una Java Server Page:

1) accesso centralizzato rispetto la JSP;
2) accesso centralizzato rispetto la Servlet.

Poiché Penelope genera automaticamente JSP attraverso il primo metodo di accesso è ad esso che faremo riferimento in questo paragrafo, rimandando per ulteriori delucidazioni sul secondo metodo di accesso all'appendice A di questa tesi.

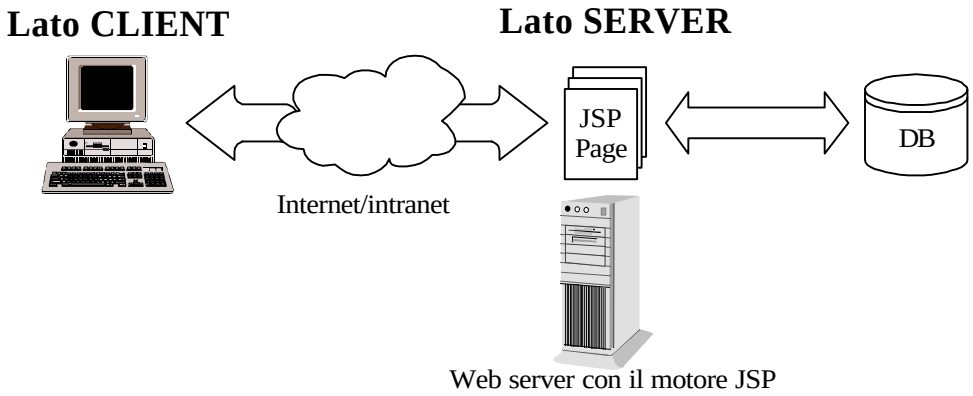


Figura 3.11: Prima architettura per l'invocazione di una pagina JSP

In questo tipo di accesso (figura 3.11) un client esegue una richiesta direttamente ad una pagina JSP, la quale "interfacciandosi" con delle risorse sul lato server, produce direttamente l'output.

La prima volta che un file JSP viene invocato, viene effettuata la sua compilazione e di tutti i componenti definiti al suo interno (codice Java, script,…), dando luogo a bytecode standard; pertanto quello che prima era script diventa codice Java compilato ed eseguito all'interno del server. Diretta conseguenza di questo meccanismo è che la prima volta che si invoca il file JSP, il sistema sarà un po' più lento del normale, ma, dalla seconda invocazione in poi, si hanno prestazioni considerevoli.

Vediamo ora come si è operato al fine di generare automaticamente, attraverso Penelope, pagine JSP.

La prima cosa che viene scritta sul file generato sono le istruzioni di import delle classi utilizzate:

```
<%@ page import="javax.servlet.http.HttpUtils" %>
<%@ page import="java.util.*" %>
<%@ page import="java.io.*" %>
<%@ page import="java.sql.*" %>
<%@ page import="java.net.*" %>
<%@ page import="DBConnectionManager" %>
```

Pattern che si occupa di gestire il pool di connessione

```
<%@ page import="java.lang.*" %>
<%@ page import="ResultSetExternal" %>
```

Classi importate solo se è presente una chiamata ad una sorgente dati esterna

Come fatto precedentemente analizziamo come, frammenti di PL, vengono tradotti da Penelope in termini di pagine JSP.

Consideriamo la seguente defnizione di page-scheme:

```
ON Sigmod

DEFINE PAGE Dynamic ProceedingsPage : "Proceedings Page"
AS  URL (<CONF.VOLUME>, <CONF.NUM>);
```

La pagina generata potrà essere chiamata attraverso il seguente URL:

http://vesuvio.dia.uniroma3.it/Sigmod/ProceedingsPage/ProceedingsPage.jsp? VOLUME=27+++&NUM=2++++

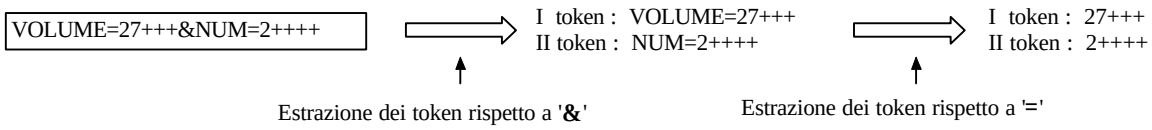
QueryString

Come prima cosa poiché la pagina generata è parametrica (il page-scheme non è unico) si effettua un controllo sulla QueryString:

```
<% String queryString = request.getQueryString();
   if (queryString == null)
   {
%>   <html>
       <head> <title>Page not available</title> </head>
       <body>
        <CENTER><H2>Wrong parameters</H2></CENTER>
       </body>
      </html>
<%
   }
   else
   ......
```

Se è nulla si visualizza un messaggio di errore che segnala che la pagina richiesta non è disponibile in quanto si sono passati dei parametri errati, altrimenti vengono catturati i parametri utilizzando la *StringTokenizer* di Java.

Mentre nelle pagine ASP si dispone di strumenti che permettono direttamente di estrarre gli elementi dalla QueryString nelle JSP ciò viene realizzato estraendo i singoli token:




```
StringTokenizer queryStringTokenizer;
queryStringTokenizer = new StringTokenizer(queryString, "&");

String par0 = (String)queryStringTokenizer.nextToken();
StringTokenizer parameter0Tokenizer = new StringTokenizer(par0, "=");
parameter0Tokenizer.nextToken();
String CONFVOLUME= (String)parameter0Tokenizer.nextToken();
try
  {
    CONFVOLUME = URLDecoder.decode(CONFVOLUME);
}catch (Exception e) {}

String par1 = (String)queryStringTokenizer.nextToken();
StringTokenizer parameter1Tokenizer = new StringTokenizer(par1, "=");
parameter1Tokenizer.nextToken();
String CONFNUM= (String)parameter1Tokenizer.nextToken();
try
  {
    CONFNUM = URLDecoder.decode(CONFNUM);
}catch (Exception e) {}
```

Si osservi che quando vengono inizializzate le variabili CONFVOLUME e CONFNUM (variabili d'istanza della pagina) viene utilizzato il metodo *decode* della classe URLDecoder. Allo stesso modo si vedrà che quando si devono generare dei link, verso pagine dinamiche parametriche, viene utilizzato il metodo *encode* della classe URLEncode. Le classi URLDecoder e URLEncode forniscono rispettivamente i metodi statici *decode* ed *encode* per la conversione di stringhe di testo in una forma che sia adatta all'uso come parte di URL. Questo formato è noto come "x-www-form-urlencoded" ed è utilizzato generalmente per codificare i dati di modulo inviati a uno script CGI.

Il metodo *encode*() converte gli spazi in segni più(+) e utilizza un segno percentuale(%) come codice di escape per codificare i caratteri speciali. I due caratteri immediatamente seguenti il segno di percentuale sono interpretati come cifre esadecimali combinate per produrre un valore ad otto bit. Il metodo *decode* fa l'operazione inversa.

Successivamente vengono definiti tre metodi :

1. Il metodo ***jspinit***, che rappresenta il momento della creazione ed istanziazione della servlet corrispondente alla JSP invocata, serve per inizializzare tutti i parametri e le variabili da utilizzare per il funzionamento. In tale metodo vengono inserite tutte le operazioni che non necessitano di esecuzione ad ogni richiesta dei client, è qui infatti che viene creato l'oggetto "pool" di tipo DBConnectionManager che consentirà la gestione del pool di connessione alla base di dati .

```
<%!
    Connection con;
    DBConnectionManager pool;

    // Method init
    public void jspInit()
    {
      try
       {
        pool = DBConnectionManager.getInstance();
        }catch(Exception e) {}
     }
%>
```

Variabili dichiarate globali in modo che possano essere utilizzate nel body della pagina

Inizializzazione del pool di connessione

2. Il metodo ***jspDestroy*** che viene invocato o quando si effettua lo *shutdown* del web server oppure quando è necessario liberare delle risorse. E' in tale metodo che avviene il rilascio di tutte le connessioni che sono state aperte per accedere al sito web generato;

```
// Method Destroy
public void jspDestroy()
{
  try
    {
    pool.release();
    }catch(Exception e) {}
}
```

Rilascio di tutte le connessioni aperte

3. Il metodo ***checkString*** invocato ogni qual volta si deve operare su attributi estratti da tabelle relazionali. La sua funzione è quella di determinare se all'interno del valore di un attributo sono presenti dei caratteri " ' " (es. "D'Amico" è il valore assunto dall'attributo *Cognome*) ed in caso affermativo di sostituirli con la codifica apici specificata in un apposito file di nome "*jdbcProperties.txt*".

```
<%
    // Method checkString
    public String checkString(String result)
    {
      int index = result.indexOf("'");
      int i= 0;
      StringBuffer resBuf = new StringBuffer(result);

      while (i<resBuf.length())
      {
        if (resBuf.charAt(i) == '\'')
        {
          resBuf.replace(i,i+1,"''");
          i++;
        }
        i++;
      }

      String out = resBuf.toString();
      return out;
    }
 %>
```

Caso in cui il db è MSACCESS

Il file *"jdbcProperties.txt"* indica come definire i parametri di connessione alla base di dati, in particolare è possibile specificare diverse variabili:

- ***codificaApici***: è la stringa che codifica gli apici

  es: MS-Access → "
  mySQL → \'

- ***codificaIsNotNull***: è la stringa che codifica la funzione Is Not Null()

  es: MS-Access → Is Not Null
  mSQL → <> Null

- ***jdbcProtocol***: è la stringa che codifica il tipo protocollo jdbc utilizzato

  es: MS-Access → jdbc:odbc:
  mSQL → jdbc:imaginary:
  Oracle → jdbc:oracle:

nel caso in cui il db non risieda sulla stessa macchina, è inoltre possibile specificare sia l'indirizzo che la porta utilizzata dal server SQL

es: jdbc:postgresql://poincare:8088/

- ***jdbcDriver***: e' la stringa che codifica il tipo driver utilizzato; si tratta in realtà della classe java che viene caricata e che implementa il collegamento JDBC.

  es: MS-Access → sun.jdbc.odbc.JdbcOdbcDriver
  mSQL → imaginary.JdbcOdbcDriver
  Oracle → oracle.jdbc.driver.OracleDriver

Riportiamo per completezza un semplice esempio :

```
#-----------------------------------------------
#       jdbcProperties.txt
#-----------------------------------------------

#MS-Access Configuration.
#Wed May 10 18:45:11 GMT+01:00 2000
codificaIsNotNull=\ Is\ Not\ Null\
jdbcProtocol=jdbc\:odbc\:
jdbcDriver=sun.jdbc.odbc.JdbcOdbcDriver
codificaApici="
#
#
#MySQL Configuration.
#Wed May 10 18:45:12 GMT+01:00 2000
password=mypassword
codificaIsNotNull=\ <>null\
login=mylogin
jdbcProtocol=jdbc\:postgresql\://hostname\:portnumber/
jdbcDriver=imaginary.JdbcOdbcDriver
codificaApici=\\'
```

Figura 3.12: Esempio file di configurazione “jdbcProperties.txt

Subito dopo i tre metodi viene aperta una connessione con la base di dati invocando il metodo *getConnection* della classe DBConnectionManager

```
<% con = pool.getConnection("sigmod"); %>
```

Poi viene scritta su file la query di livello 0 (nel nostro esempio parametrica poiché la funzione URL è definita sulla base di due valori non costanti).

```
<% Statement rstStmt = con.createStatement();
    String rstQuery = "Select Distinct CONFERENCES.VOLUME, CONFERENCES.NUM,
                                CONFERENCES.CONFDATE, CONFERENCES.CONFYEAR,
                                CONFERENCES.CONFNAME, CONFERENCES.LOCATION,
                                CONFERENCES.STATE, RECORDISSUES.MONTH,
                                RECORDISSUES.YEAR
                      From CONFERENCES, RECORDISSUES
                      Where CONFERENCES.VOLUME=RECORDISSUES.VOLUME AND
                            CONFERENCES.NUM=RECORDISSUES.NUM AND
                            CONFERENCES.VOLUME =  "'+CONFVOLUME+'" AND
                            CONFERENCES.NUM =  "'+CONFNUM+'"";   %>
```

Si noti che i parametri, CONFVOLUME e CONFNUM, rispetto ai quali si esegue la query sono quelli ottenuti precedentemente estratti dalla QueryString.

Viene scritto come eseguire la query e come controllare se il cursore è nullo; se è nullo viene visualizzato un messaggio di errore che segnala che la pagina richiesta non è disponibile in quanto si sono passati dei parametri errati altrimenti si scrive come inizializzare tante stringhe quanti sono gli attributi ritornati per ogni tupla del cursore. Il nome delle stringhe viene assegnato concatenando il nome del cursore con il nome dell'attributo.

```
<%   if (rstStmt.execute(rstQuery))
     {
      ResultSet rst = rstStmt.executeQuery (rstQuery);

      if (rst.next()){

     String rstVOLUME = rst.getString("VOLUME");
     String rstNUM = rst.getString("NUM");
     String rstCONFDATE = rst.getString("CONFDATE");
     String rstCONFYEAR = rst.getString("CONFYEAR");
     String rstCONFNAME = rst.getString("CONFNAME");
     String rstLOCATION = rst.getString("LOCATION");
     String rstSTATE = rst.getString("STATE");
     String rstMONTH = rst.getString("MONTH");
     String rstYEAR = rst.getString("YEAR");
 %>
```

Immaginiamo ora che nel PL sia presente un attributo composto, ad esempio:

```
sectionList : LIST-OF (
                        sectionName    : TEXT = <SECTIONS.SECTION>,
                                OFFSET(<SECTIONS.SECTIONPOSITION>);
                      );
```

Questo sul file JSP generato equivale a scrivere la query (punto 1), il controllo se l'esecuzione della query porta ad un cursore nullo (punto 2), come eseguire la query (punto 3), un ciclo while che termina quando non ci sono più tuple nel cursore (punto 4), tante stringhe quanti sono gli attributi ritornati dalla query (punto 5,6,7,8). Viene poi scritto come stampare l'attributo `sectionName` (punto 9). Infine si chiudono

i cursori (punto 10 e 11) .

```
1. <%  Statement sectionListStmt = con.createStatement();
       String sectionListQuery = "Select  Distinct ARTICLES.VOLUME, ARTICLES.NUM, ARTICLES.SECTION,
                                          MIN(POSITION) AS SECTIONPOSITION
                                  From ARTICLES
                                  Where ARTICLES.VOLUME = "'+rstVOLUME+'" AND
                                          ARTICLES.NUM = "'+rstNUM+'"
                                  Group By VOLUME, NUM, SECTION Order By MIN(POSITION)";
2.  if (sectionListStmt.execute(sectionListQuery))
    {
3.  ResultSet sectionList= sectionListStmt.executeQuery (sectionListQuery);

4.  while(sectionList.next()) {
5.       String sectionListVOLUME = sectionList.getString("VOLUME");
6.       String sectionListNUM = sectionList.getString("NUM");
7.       String sectionListSECTION = sectionList.getString("SECTION");
8.       String sectionListSECTIONPOSITION = sectionList.getString("SECTIONPOSITION");
    %>

9.   <tr><td><strong><% if (sectionListSECTION!= null) out.print(sectionListSECTION); %></strong><td></tr>

    <%   }/*end while sectionList*/
10.       sectionList.close();
11.      sectionListStmt.close();
     }/*end if empty cursor sectionList*/ %>
```

Vediamo ora come vengono tradotti gli attributo LINK-TO. Supponiamo di aver definito:

```
Toindex  : LINK-TO IndexTermsPage (
             URL(<ARTICLECODE>);
             index : TEXT = "Abstract, Index Terms and Cathegories";
                                );
```

Penelope codifica questo scrivendo sul file JSP:

Parametro d'istanza della pagina

```
<A HREF="../IndexTermsPage/IndexTermsPage.jsp?
        ARTICLECODE=<% out.print(URLEncoder.encode(articlesARTICLECODE)); %>">
        Abstract, Index Terms and Cathegories) </A>
```

Ancora

Poiché l'URL è definito sulla base di un valore variabile sulla base di quanto già detto nel paragrafo 2.4.1 il nome della pagina sarà proprio il nome del page-scheme linkato.

Alla fine della pagina JSP viene chiuso il cursore di livello 0 e viene liberata la connessione:

```
<%     rst.close();
       rstStmt.close();
       ....
}/*end if empty cursor rst*/
  .....
    pool.freeConnection("sigmod",con);
   } /*end if check QueryString*/ %>
```

3.7 Generazione automatica di programmi

Il nuovo prototipo Penelope è in grado di generare programmi Java che una volta compilati ed eseguiti permettono di ottenere il sito web statico, in formato HTML oppure in formato XML.

In figura 3.13. riportiamo il diagramma della generica classe generata.

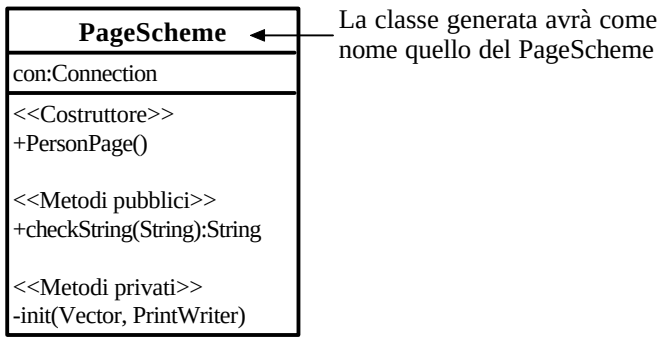


Figura 3.13: Diagramma della generica classe Java generata

I file Java generati hanno lo stesso nome del page-scheme e all'interno della classe sono presenti due metodi:

1. Il metodo ***checkString*** descritto nel paragrafo precedente;
2. Il metodo ***init*** che si occupa di eseguire le query che permettono di estrarre tutti e solo gli attributi definiti nel page-scheme e di scrivere su file tutti i tag e le informazioni che compongono la pagina web che si sta generando. Il metodo prende in input due parametri:

- un vettore *parametri* che contiene le condizioni d'istanza della pagina necessarie per ottenere il cursore che consente di estrarre gli attributi di livello 0. Nel caso in cui il page-scheme sia unico, e quindi verrà generata una sola pagina web (HTML o XML), tale vettore non conterrà elementi (sarà *null*);
- un oggetto di tipo *PrintWriter* che attraverso il metodo *write* permette di stampare sul file (rispetto al quale è stato inizializzato) tutte le informazioni.

Nel *main* del file Java si effettua la connessione alla base di dati, si esegue, nel caso di pagina non unica, la query di livello zero (non parametrica), si scandisce,mediante un ciclo, il cursore ottenuto; all'interno del ciclo si memorizzano, nel vettore di input del metodo *init*, quelli che sono i valori d'istanza delle pagine che devono essere generate, si inizializza l'oggetto "out" di tipo *PrintWriter* che viene passato al metodo *init* ed infine viene invocato il metodo *init*.

L'oggetto *PrintWriter* viene inizializzato con un *FileOutputStream* che crea l'*OutputStream* utilizzato per scrivere su file. Al *FileOutputStream* viene passato il percorso del file che si vuole creare. Il nome del file che si genera sarà costante se il page-scheme è unico, oppure sarà di volta in volta l'elemento corrente del vettore *parametri*.

Consideriamo ad esempio il seguente page-scheme:

```
DEFINE PAGE PersonPage : <Name>
AS URL(<Person.Name>);
   Name : TEXT = <Name>;
   Photo : IMAGE = <Photo>;
   Position : TEXT = <Position>;
   Email : TEXT = <email>;
   Phone : TEXT = <Phone>;
   GroupList : LIST-OF (ToGroup : LINK-TO ResearchGroupPage (

                                        URL(<PersonGroup.GroupName>);
                                        LabelGroup : TEXT = <GroupName>;
                                        );
                          );
   CourseList : LIST-OF (ToCourse : LINK-TO CoursePage (
                                     URL(<CourseName>);
                                     LabelCourse : TEXT = <CourseName>;
                                   );
```

```
                          );
USING Person,
      PersonGroup : (SELECT Person.Name, PersonInGroup.GroupName
                      FROM Person, PersonInGroup
                      WHERE Person.Name = PersonInGroup.Name) DISTINCT,
      PersonCourses : ( SELECT Person.Name, Course.CourseName
                         FROM Person, Course
                         WHERE Course.Instructor = Person.Name )
DISTINCT
END
```

Il file Java che viene generato da Penelope è il seguente:

```
//  ************************************************************************
//  Page generated by PENELOPE.
//  Copyright (c) 2000 Araneus Group and University 'Roma Tre', Rome, ITALY
//  All rights reserved.
//  ************************************************************************

import java.util.*;
import java.io.*;
import java.sql.*;
import java.lang.*;
import java.net.*;


class PersonPage
{
  Connection con;

  public PersonPage()
  {}

  // Method checkString
  public String checkString(String result)
  {
    int index = result.indexOf("'");
    int i= 0;
    StringBuffer resBuf = new StringBuffer(result);
    while (i<resBuf.length())
     {
       if (resBuf.charAt(i) == '\'')
       {
         resBuf.replace(i,i+1,"''");
         i++;

   }
       i++;
     }
    String out = resBuf.toString();
    return out;
  }


  private void init(Vector parametri, PrintWriter out)
  {
    String PersonName = (String)parametri.elementAt(0);
    try
    {

    // Query di Livello 0
      Statement rstStmt = con.createStatement();
      String rstQuery = "Select Person.Phone, Person.email, Person.Position,
                         Person.Photo, Person.Name
                         From Person Where Person.Name = '"+ checkString(PersonName) +"'";
```

```
      if (rstStmt.execute(rstQuery))
       {
         ResultSet rst = rstStmt.executeQuery (rstQuery);
         // Si posiziona sulla prima tupla del cursore. Se non si mette da errore perche'
         // accede ad una tupla del cursore che non esiste
         if (rst.next()){

        String rstPhone = rst.getString("Phone");
        String rstemail = rst.getString("email");
        String rstPosition = rst.getString("Position");
        String rstPhoto = rst.getString("Photo");
        String rstName = rst.getString("Name");

out.println("<HTML>\n<!--\nPage generated by PENELOPE.\nCopyright (c) 1997 Araneus Group
and University 'Roma Tre', Rome, ITALY\nAll rights reserved.\n-->\n<TITLE>");

 if (rstName!= null) out.print(rstName);

out.println("</TITLE><BODY>");
out.println(" <P>");
out.println("");

 if (rstName!= null) out.print(rstName);

out.println("</P>");
out.println(" <P>");
out.println("");
out.print("<IMG ");
out.print("");
out.print(" SRC=\"");

 if (rstPhoto!= null) out.print(rstPhoto);

out.print("\"");
out.println(">");
out.println("</P>");
out.println(" <P>");
out.println("");

 if (rstPosition!= null) out.print(rstPosition);

out.println("</P>");
out.println(" <P>");
out.println("");

 if (rstemail!= null) out.print(rstemail);
out.println("</P>");
out.println(" <P>");
out.println("");

 if (rstPhone!= null) out.print(rstPhone);

out.println("</P>");
out.println(" <UL>");
out.println(" ");
out.println("    ");

Statement GroupListStmt = con.createStatement();
String GroupListQuery = "Select  Distinct Person.Name, PersonInGroup.GroupName
                         From Person, PersonInGroup Where Person.Name = PersonInGroup.Name
                              AND Person.Name = '"+ checkString(rstName) +"'";

   if (GroupListStmt.execute(GroupListQuery))
   {

 ResultSet GroupList= GroupListStmt.executeQuery (GroupListQuery);

   while(GroupList.next()) {
         String GroupListName = GroupList.getString("Name");
         String GroupListGroupName = GroupList.getString("GroupName");
```

```
          out.println("    <P>");
          out.print(" <A HREF=\"");
          out.print("../ResearchGroupPage/");
          out.print(GroupListGroupName.trim() + ".html");
          out.println("\"");
          out.print(">");
          out.print("");
          out.print("    ");
          out.print("<P>");
          out.print("");
           if (GroupListGroupName!= null) out.print(GroupListGroupName);
          out.print("</P>");
          out.print("</A>");
          out.println("</P>");
          out.println("\n    ");
   }/*end while GroupList*/
   GroupList.close();
   GroupListStmt.close();
}/*end if empty cursor GroupList*/

out.println(" ");
out.println(" </UL>");
out.println(" <UL>");
out.println(" ");
out.println("    ");

  Statement CourseListStmt = con.createStatement();
  String CourseListQuery = "Select  Distinct Person.Name, Course.CourseName
                           From Person, Course
                           Where Course.Instructor = Person.Name AND
                                 Person.Name = '"+ checkString(rstName) +"'";

   if (CourseListStmt.execute(CourseListQuery))
   {
   ResultSet CourseList= CourseListStmt.executeQuery (CourseListQuery);

   while(CourseList.next()) {
         String CourseListName = CourseList.getString("Name");
         String CourseListCourseName = CourseList.getString("CourseName");

          out.println("    <P>");
          out.print(" <A HREF=\"");
          out.print("../CoursePage/");
          out.print(CourseListCourseName.trim() + ".html");
          out.println("\"");
          out.print(">");
          out.print("");
          out.print("    ");
          out.print("<P>");
          out.print("");

          if (CourseListCourseName!= null) out.print(CourseListCourseName);

          out.print("</P>");
          out.print("</A>");
          out.println("</P>");
          out.println("\n    ");
   }/*end while CourseList*/
   CourseList.close();
   CourseListStmt.close();
}/*end if empty cursor CourseList*/

out.println(" ");


out.println(" </UL>");
out.println("</BODY></HTML>");

       }/*end while rst*/
       rst.close();
```

```
      rstStmt.close();
    }/*end if empty cursor rst*/
  } catch (SQLException e)  { out.println(e.getMessage()); }
  out.close();
  return;
}// end Method init


public static void main(String[] args)
{
  try
  {
    PersonPage personpage = new PersonPage();
    try
     {
        Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
     }
     catch (ClassNotFoundException ex2)
     {
        System.err.println("Connection at database failed. Error 1");
        System.exit(-1);
     }
     catch (Exception ex)
     {
        System.err.println("Connection at database failed. Error 2");
        ex.printStackTrace();
        System.exit(-1);
     }
     try
     {
        personpage.con = DriverManager.getConnection("jdbc:odbc:department");
     }
     catch (SQLException e)
     {
        System.err.println("Connection at database failed.. Error 3");
        System.err.println("The source date ODBC not exist.");
        System.exit(-1);
     }
     catch (Exception ex)
     {
        System.err.println("Connection at database failed. Error 4");
        ex.printStackTrace();
        System.exit(-1);
     }
      Statement rstStmt = (personpage.con).createStatement();
      String rstQuery = "Select Person.Phone, Person.email, Person.Position,
                                Person.Photo, Person.Name
                         From Person";

      if (rstStmt.execute(rstQuery))
      {
        ResultSet rst= rstStmt.executeQuery (rstQuery);

        while(rst.next())
        {
          Vector parametri = new Vector();
           parametri.addElement(rst.getString("Name"));
          try
          {
            PrintWriter out = new PrintWriter(new FileOutputStream("PersonPage\\" +
                              ((String)parametri.elementAt(0)).trim() + ".html"));
            personpage.init(parametri, out);
          }catch(IOException ioe) { System.out.println("IOException writing
                                    file:\n"+ioe); }
        }
      }
  }catch (Exception e)  { System.out.println(e.getMessage()); }
}// end main
}// end class personpage
```