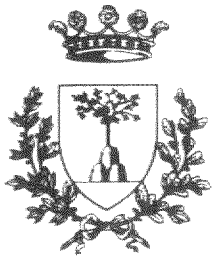

Università degli Studi di Ferrara

Facoltà di Ingegneria

Corso di Laurea in Ingegneria Elettronica

Interfacce Intelligenti per il Recupero di Informazioni su Internet

Tesi di Laurea di:
Andrea Vicentini

Relatore:
Chiar.mo Prof. Ing. Paola Mello

Correlatori:
Dott. Ing. Rita Cucchiara
Dott. Claudio Benvenuti

Anno Accademico 1996/97

Indice

Indice

Elenco delle figure

Elenco delle figure

Introduzione

Nel panorama delle applicazioni informatiche i servizi di accesso alle informazioni geograficamente distribuite stanno acquistando una diffusione ed un ruolo sempre più importante. L'esempio notevole in questo senso è rappresentato dall'insieme di servizi di rete offerti da Internet: posta elettronica, liste di utenti, fornitura di prodotti commerciali di ampia diffusione e servizi alle imprese.

È proprio in quest'ultimo ambito che si configura il lavoro presentato in questa tesi, svolto in collaborazione con Talete, società del gruppo Zuffellato, con la quale il Dipartimento di Ingegneria ha una convenzione nell'ambito dell'iniziativa comunitaria "Adapt (priorità Adapt Bis – Building the Information Society)". Il progetto ha titolo "*Servizi multimediali interattivi per lo sviluppo del commercio internazionale: la PMI entra nella società dell'informazione*".

Tale progetto comporta la realizzazione di un servizio Internet costituito da un sistema informativo il quale sappia rendere fruibili da parte degli utilizzatori il più vasto e completo insieme di informazioni (tra quelle disponibili in Internet nell'ambito dello sviluppo del commercio internazionale) che sia stato possibile raccogliere e catalogare. Tali informazioni serviranno come supporto decisionale per la Piccola e Media Impresa (PMI), destinataria del progetto comunitario.

Nell'ottica di dare una effettiva realizzazione al sistema informativo obiettivo del progetto Adapt, il lavoro svolto in questa tesi ha comportato da un lato lo studio delle tecniche di Intelligenza Artificiale volte a fornire un servizio migliore, cioè più facilmente ed efficacemente fruibile da parte dell'utente; dall'altro ha comportato il completo sviluppo del sistema informativo, che risiede nel sito Internet di Talete.

Il compito principale del sistema è sfruttare la propria base di conoscenza per recuperare informazioni distribuite su Internet. Nello sviluppo del progetto sono state applicate tecniche di Intelligenza Artificiale nell'interazione con l'utente e nella categorizzazione e strutturazione della base di conoscenza.

Il progetto prevede una interfaccia utente efficiente e flessibile, orientata

ad utenti non esperti, ai quali è consentito porre domande al sistema informativo interloquendo formalmente con un operatore artificiale (il gestore della base di conoscenza) e con un operatore umano (l'"esperto in linea"), capace di supplire alle eventuali carenze del sistema artificiale.

Il sistema artificiale è in grado anche di apprendere dall'interazione con l'esperto umano: infatti le risposte dell'esperto vengono integrate nella base di conoscenza del sistema.

Lo sviluppo della tesi si è avvalso di tecniche di informatica avanzata, come il linguaggio di programmazione Java e degli strumenti tecnologici disponibili oggi per la realizzazione di applicativi il cui funzionamento sia intrinsecamente legato all'interazione con la rete.

In questo ambito è stata svolta una iniziale attività di reperimento ed analisi di prestazioni dei programmi, già presenti e disponibili in Internet, che avrebbero potuto essere utili per la realizzazione del servizio. Tali programmi sono stati in parte adattati ed in parte totalmente rimpiazzati da software originale, specificatamente progettato e sviluppato in questa tesi.

Il risultato del lavoro condotto è un prototipo funzionante e completo in ogni sua parte essenziale; è previsto un proseguimento dell'attività per integrare altre funzionalità innovative (come ad esempio una sofisticata modellizzazione dell'utente) tipiche dell'Intelligenza Artificiale, che sono state studiate e previste nello sviluppo di questa tesi.

Struttura del lavoro

La presente tesi è idealmente suddivisa in due parti: nella prima sono stati analizzati i prodotti disponibili ed impiegabili per la realizzazione del sistema informativo; nella seconda viene dettagliatamente analizzato il lavoro svolto, consistente in una fase di analisi di progetto ed una di sviluppo.

Il capitolo 1 definisce ed analizza diversi sistemi per il recupero delle informazioni in rete, comparandone ove possibile le prestazioni e le capacità.

Il capitolo 2 studia il funzionamento e le caratteristiche dei motori di ricerca, particolari sistemi per il recupero delle informazioni tipici di Internet.

Il capitolo 3 analizza vari programmi di *spider*, che copiano informazioni da una macchina remota alla macchina locale.

Il capitolo 4 è il capitolo in cui viene introdotto il linguaggio Java, che rappresenta la tecnologia con cui l'intero sistema informativo presentato in questo lavoro è stato realizzato. Java viene analizzato da un punto di vista funzionale (comparandolo con i linguaggi affini) e puramente tecnico (descrivendo le particolarità che lo contraddistinguono). In questo stesso fondamentale capitolo vengono sinteticamente descritti ed analizzati gli altri

strumenti tecnologici disponibili per sviluppare programmi che interagiscano con la rete: HTML, JavaScript ed i programmi CGI.

Nella seconda parte, il capitolo 5 descrive le specifiche che l'applicazione prodotta dovrà soddisfare. Dapprima vengono esaminati gli obiettivi del progetto europeo Adapt, successivamente si definiscono le specifiche progettuali: dalla scomposizione del sistema in blocchi funzionali all'esaustiva descrizione del modo di funzionamento di ognuno. Dopo aver chiarito il comportamento dei vari blocchi e le relazioni che intercorrono fra loro, vengono scelti i componenti (fra quelli esaminati nella prima parte della tesi) con cui realizzare le varie parti del progetto.

Nel capitolo 6 viene descritto l'adattamento di programmi preesistenti, reso necessario per una loro fattiva integrazione nell'architettura proposta in sede progettuale.

Il capitolo 7 contiene una approfondita descrizione e discussione delle parti più significative del codice sviluppato in questa tesi.

Il capitolo 8 elenca una serie di possibili future estensioni da realizzare a questo sistema informativo, nell'ottica di renderlo maggiormente utile ed efficacemente utilizzabile da parte degli utenti.

In appendice si trovano i file di configurazione del sistema, una parte significativa del codice sorgente sviluppato ed un allegato che esemplifica il tipo di documentazione tecnica sul programma automaticamente producibile dall'ambiente Java.

Parte 1

Capitolo 1

Sistemi di recupero delle informazioni

Con il termine "Recupero delle Informazioni" (o "Information Retrieval") si intende il processo attraverso il quale un sistema di elaborazione può convertire una richiesta di informazioni in un'utile collezione di riferimenti. Un sistema di recupero dell'informazione è in generale capace di estrarre e strutturare dati ad elevato contenuto informativo a partire da grandi quantità di sorgenti di informazioni, anche non strutturate, e di qualunque tipo (testi, immagini, suoni, video...)[1].

Oggetto di questa tesi sarà lo studio e lo sviluppo di un sistema per il recupero di informazioni di tipo testuale, quindi l'attenzione sarà ristretta a questa tipologia di sistemi.

I sistemi per il recupero di informazioni testuali adottano un approccio ormai consolidato, basato sulle seguenti due funzioni:

- *indicizzazione*: a partire dalle informazioni testuali (che compongono la base di dati a disposizione del sistema di recupero delle informazioni) viene generato un *indice* delle informazioni medesime. L'indice è una struttura che consente un immediato recupero delle informazioni indicizzate in base al contenuto. Dal punto di vista della tecnologia informatica, le strutture dati che vengono più spesso utilizzate sono le tabelle hash e gli alberi bilanciati[18].

- *ricerca*: estrae un elenco di riferimenti dall'indice precedentemente creato in base ad una *chiave* di ricerca.
  Una chiave di ricerca è in generale un'espressione che contiene una visione parziale dell'informazione che si desidera estrarre. Il programma che esegue la ricerca cercherà una corrispondenza fra l'espressione ed

i testi contenuti nella base di dati a disposizione del sistema, generando un riferimento per ogni corrispondenza trovata. Quest'opera di generazione dei riferimenti è resa possibile e veloce dalla preliminare indicizzazione.
Per ogni riferimento viene generalmente espressa una valutazione numerica (*rank*) che rappresenta una stima del grado di interesse dell'utente nei confronti del documento riferito.

I sistemi per il recupero di informazioni dotati di queste due sole funzioni sono perciò in grado di operare esclusivamente con una base di dati che risieda sulla memoria di massa privata (generalmente il disco rigido) della macchina su cui sono in esecuzione.

Nel caso in cui le informazioni che compongono la base di dati siano *non locali*, ossia disponibili esclusivamente sulla rete Internet, sarà necessario dotare il sistema di una ulteriore funzione:

- "recupero" (*spidering*): trasferisce il contenuto delle informazioni che compongono la base di dati dal sito in cui si trovano nel disco rigido della macchina locale (per maggiori dettagli, vedere il capitolo 3).

Nei paragrafi che seguono vengono presentati alcuni prodotti esaminati nell'ambito degli strumenti di recupero dell'informazione. Tra tutti quelli disponibili nel Web[1], quelli presentati sono gli unici che sembrano utilizzabili nel contesto dell'applicazione da sviluppare.

Tutti i pacchetti offrono eccellenti prestazioni dal punto di vista della velocità di ricerca dei documenti mediante chiave, mentre piuttosto dissimili appaiono le loro caratteristiche per quanto riguarda la velocità di generazione dell'indice e lo spazio da esso occupato sul disco rigido.

Infine, è interessante osservare come ognuno dei programmi di questo capitolo sia stato concepito principalmente come indicizzatore di documenti in formato HTML (vedere il paragrafo 4.3.1), dal momento che tutti prevedono meccanismi di interpretazione della struttura che compone queste particolari informazioni testuali.

Prima di proseguire nel dettaglio di tali prodotti, è necessario premettere alcune definizioni di carattere generale, per meglio comprendere qual è l'ambito in cui questi programmi funzionano.

[1]Per avere un elenco più dettagliato dei sistemi per il recupero dell'informazione disponibili si consulti la categoria di Yahoo!

Computers and Internet: Internet: World Wide Web: Databases and Searching

Il **World Wide Web**[21](in breve, WWW) è un sistema informativo che consente di usufruire delle informazioni immagazzinate in diversi computer attraverso riferimenti ipertestuali. L'uso del servizio avviene mediante una connessione di tipo *client-server* attraverso la rete, con il protocollo HTTP (HyperText Transfer Protocol).
È detta *server* del servizio WWW la macchina su cui risiedono i file sorgenti di informazioni, che possono essere di qualunque tipo: immagini, suoni, Applet (vedere il capitolo 4) ed in particolare HTML (HyperText Markup Language: vedere il paragrafo 4.3.1). Il *server* è in grado di inviarli al *client* che ne faccia richiesta per una elaborazione.
È detta *client* la macchina dell'utente che desidera avere le informazioni immagazzinate nelle macchine *server*. I programmi che consentono lo scambio di codeste informazioni fra *client* e *server* sono detti *browser*.

Con il termine **browser**[21] si indicano programmi adibiti alla connessione e all'interscambio di informazioni fra le macchine *client* (su cui sono in esecuzione) ed i *server* del servizio WWW (a cui inviano le richieste di informazioni).
Esistono molti programmi di questo tipo, sia di pubblico dominio che commerciali, sia con interfaccia grafica che a caratteri. I più sono in grado di connettersi non solo con server di servizio WWW, bensì anche con tutti gli altri servizi Internet (ftp, telnet, Usenet News ...).

1.1 ht://Dig

ht://Dig è un programma sviluppato alla San Diego State University da Andrew Sherpbier, disponibile in rete gratuitamente dal sito:

```
http://htdig.sdsu.edu/
```

Una caratteristica saliente di **ht://Dig** rispetto agli altri prodotti esaminati in questo capitolo è sicuramente il fatto di offrire un sistema completo per il recupero delle informazioni, il quale si incarica cioè dello *spidering*, dell'indicizzazione e della ricerca.
Il pacchetto software si compone di quattro parti, ognuna delle quali si occupa di un singolo processo di elaborazione dell'informazione:

- **htdig**: è il programma che recupera le informazioni da indicizzare, prelevandole da uno o più siti remoti, e le copia in un direttorio sul disco rigido locale, uno distinto per ogni sito esaminato; è inoltre possibile anche specificare una coppia *userid:password* per collegarsi a siti protetti da accessi non autorizzati.

htdig è la componente del pacchetto che lo differenzia dagli altri prodotti esaminati in questo capitolo: essi sono infatti sistemi per il recupero di informazioni *locali*, mentre **ht://Dig** è un sistema più completo, in grado di indicizzare informazioni non locali.
La componente **htdig** ha un funzionamento molto simile a quello di un programma di *spider*, ma è assolutamente unico nel suo genere in quanto ha un comportamento che non può essere disgiunto dall'interazione con le altre componenti del pacchetto: a differenza degli *spider* tradizionali, infatti, **htdig** non produce localmente una copia delle informazioni recuperate dalla rete; esse vengono invece compattate in due file, che costituiranno parte dell'indice finale.

- **htmerge**: è il programma di indicizzazione. Elabora quanto prodotto da **htdig** e genera altri due file, che completano l'indice.
  L'indice può essere generato incrementalmente: verranno cioè elaborati solo i testi che risultino modificati dalla precedente indicizzazione, per ridurre i tempi di elaborazione.
  htmerge può anche unire diversi indici in uno unico (*merging*).

- **htfuzzy**: è un programma che rielabora l'indice stesso per consentire ricerche secondo algoritmi particolari: per assonanza, per sinonimi, con "stemming"[37][33] (ricerca per parole indipendentemente dalla desinenza), e con eliminazione delle "stop words" (parole troppo comuni per portare significato semantico all'informazione estratta). Chiaramente, poter disporre di queste funzionalità ha un costo, che si paga in termini dell'aumento delle dimensioni dell'indice e del tempo impiegato a generarlo.

- **htsearch**: è il programma di ricerca.
  Dal punto di vista della tecnologia informatica, htsearch è un programma CGI. I **programmi CGI**[21] (Common Gateway Interface) sono programmi in esecuzione su macchine server del servizio WWW, in grado di interagire con macchine client ricevendo dati in ingresso e fornendo un risultato in formato HTML.
  Nel suo funzionamento tradizionale, htsearch si interfaccia con un *browser*, costruendo come risultato della ricerca un testo HTML con l'elenco dei riferimenti alle informazioni trovate; modificando il file di configurazione di **ht://Dig** è comunque possibile determinare un formato del tutto arbitrario a questo risultato.

Il programma è disponibile sotto forma di sorgenti C++ per piattaforme Unix, ed è utilizzabile sotto la licenza d'uso generale della GNU (GNU General Public License).

Rispetto agli altri strumenti esaminati in questa sezione, **ht://Dig** è il peggiore nella fase di indicizzazione: la generazione dell'indice è infatti un processo estremamente lento e l'indice risultante ha dimensioni simili a quelle dei dati originali (in definitiva, si occuperebbe il medesimo spazio se li si trasferisse completamente sul disco rigido). La velocità di ricerca è tra le migliori al pari di **SWISH** (vedere il paragrafo 1.2). La completezza dei risultati (intesa come quanti documenti sono stati estratti in seguito ad una richiesta) è la migliore.

1.2 SWISH

SWISH (acronimo di Simple Web Indexing System for Humans) è un programma scritto e prodotto da Kevin Hughes, disponibile gratuitamente dal sito:

`http://www.eit.com/software/swish/`

SWISH è un programma che dispone di due modalità di funzionamento, corrispondenti alle due funzioni che caratterizzano un sistema per il recupero delle informazioni:

- generazione dell'indice partendo da informazioni contenute nel disco rigido locale (non è prevista cioè alcuna funzione di *spider*).
  In questa fase, il comportamento di **SWISH** è determinato dalle informazioni presenti in un file di configurazione, attraverso le quali è possibile decidere, ad esempio, quali direttori del proprio disco rigido indicizzare, quali file escludere, quali "stop words" utilizzare.
  Come **ht://Dig**, anche **SWISH** dispone di una funzione per unire più indici in uno unico.

- estrazione dei riferimenti ai testi indicizzati usando chiavi di ricerca di buona complessità (cioè piuttosto espressive).
  In questa fase, **SWISH** è sensibile a varie opzioni, impostate dalla linea di comando con cui è eseguito. In particolare, si possono specificare le parti dei testi HTML indicizzati a cui la ricerca andrà limitata; possono dunque essere estratti documenti in cui la chiave di ricerca compaia solo nel titolo, nelle intestazioni dei capitoli, nelle frasi in grossetto, nei commenti, oltre che, ovviamente, in qualunque parte del documento.

La chiave con cui avviene l'estrazione è in generale un'espressione logica, basata sull'uso degli operatori `and`, `or`, `not`, `(` e `)`. È possibile anche eseguire ricerche di parole indipendentemente dalla desinenza ("stemming"), terminando la radice della parola con `*`.

Sono numerosissimi i siti web che adottano **SWISH** come strumento per la ricerca di informazioni al proprio interno: si possono citare ad esempio Cineca[2] ed NCSA[3].

SWISH è disponibile in forma di sorgenti C (*non* standard ANSI). È stato scritto e funziona correttamente sotto qualunque piattaforma Unix, ma può essere facilmente portato anche in ambiente Dos/Windows.

SWISH non è dotato di algoritmi sofisticati per la ricerca come **ht://Dig**, e nemmeno è pensato per manipolare enormi moli di dati come **Personal AltaVista** (vedere il paragrafo 1.3) o un motore di ricerca (vedere il capitolo 2). Esso offre altresì ottime prestazioni: comparandolo con gli altri prodotti esaminati in questo capitolo, è infatti il programma che genera gli indici più rapidamente e delle dimensioni minori.

Rispetto agli altri programmi considerati, **SWISH** è l'unico che non è in grado di interfacciarsi autonomamente con un *browser*, nel senso che la funzione di ricerca non costruisce un testo HTML con l'elenco dei riferimenti estratti dall'indice, non è in definitiva un programma CGI.

Per introdurre questa funzionalità è stato scritto un programma CGI apposito, di nome **wwwwais**, ottenibile gratuitamente dallo stesso sito di SWISH:

`http://www.eit.com/software/wwwwais/wwwwais.html`

Esso ha il compito di rendere comprensibili a **SWISH** le richieste dell'utente, inserendo poi in un documento HTML i riferimenti prodotti come risultato della ricerca.

SWISH è la versione semplificata e ridotta di **Isite**, che è un programma commercializzato dalla CNIDR, ottenibile dal sito Internet

`http://vinca.cnidr.org/software/Isite/Isite.html`

Questo sistema è degno di nota per motivi storici essendo un applicativo che ancora adotta il protocollo di indicizzazione e ricerca Z39.50, che fu lo standard dell'ormai estinto servizio WAIS. In effetti, **Isite** è quanto resta di

[2] `http://www.cineca.it`
[3] `http://www.ncsa.com`

un WAISServer.
Il servizio WAIS[24] è sostanzialmente la realizzazione di un sistema per il recupero delle informazioni distribuite in rete:

- ogni macchina server del servizio WAIS mantiene un indice locale delle informazioni presenti su di essa;
- i clienti del servizio estraggono i riferimenti dagli indici dei server inviando la chiave di ricerca con il protocollo di trasmissione in rete Z39.50;

In questo modo, tutte le informazioni indicizzate nei server del servizio WAIS sono disponibili a chiunque senza necessità di *spider*.

1.3 Personal AltaVista

Personal AltaVista è la versione per PC dell'omonimo motore di ricerca prodotto da Digital; è utilizzabile in versione di valutazione:

```
http://204.123.2.99/search/searchpx97_w95nti_intl.exe
```

Personal AltaVista è un sistema per il recupero delle informazioni che ha origine dal più famoso motore di ricerca per Internet **AltaVista** (vedere il paragrafo 2.1.2), il quale è stato modificato per funzionare localmente su una macchina. Per questo motivo **Personal AltaVista** dispone della tecnologia di indicizzazione e ricerca più sofisticata rispetto a tutti gli altri presi in considerazione in questo capitolo.

Al pari di **SWISH**, dispone anch'esso di due sole funzioni, realizzate dai seguenti programmi:

- programma di indicizzazione: genera l'indice basandosi sul contenuto di documenti presenti localmente alla macchina. L'interazione con l'utente avviene mediante l'interfaccia grafica di Windows. È consentita la configurazione della periodicità con la quale sarà ripetuta l'opera di indicizzazione e su quali documenti (la base di dati) verterà.
  È possibile scegliere in maniera molto flessibile quali tipi di file indicizzare e quale struttura di direttori esplorare. Non è invece prevista la possibilità di creare in maniera semplice indici distinti in corrispondenza ad indicizzazioni di zone diverse del disco rigido, nè di fare operazioni di *merging*.

- programma di ricerca (chiamato "query dispatcher"): è un programma CGI che funziona quindi interagendo con l'utente mediante l'uso di un *browser*; presenta a video la medesima interfaccia del motore di ricerca AltaVista (vedere il paragrafo 2.1.2).
  Analogamente ad esso, esistono due modalità di ricerca:
  - ricerca semplice, in cui sono estratti i riferimenti ai documenti che contengono almeno una delle parole digitate come chiave di ricerca;
  - ricerca complessa, in cui i riferimenti sono estratti adottando come chiave di ricerca una espressione logica, a cui sono aggiunti gli operatori `near` (impone che le parole si trovino vicine nel testo) e `adj` (impone l'adiacenza fra le parole).

Rispetto agli altri pacchetti di questo capitolo, Personal AltaVista offre un'ulteriore importante funzionalità: l'accesso diretto alla rete, ed in particolare alle informazioni residenti nell'indice principale di AltaVista a Palo Alto. All'atto della richiesta della chiave di ricerca, l'utente può infatti scegliere se limitare la sua ricerca all'indice presente sulla sua macchina o estenderla all'indice principale (che mantiene informazioni su tutti i documenti presenti in rete).

Da ultimo, si osserva come le varie componenti del programma si scambino dati non direttamente, bensí appoggiandosi al software di rete (facendo riferimento all'indirizzo di *loop back* 127.0.0.1, che identifica una connessione con la macchina locale). Questo implica che **Personal AltaVista**, a differenza ad esempio di **SWISH**, non è utilizzabile in assenza di una scheda di rete.

Capitolo 2

Motori di Ricerca

Nel contesto del World Wide Web, il termine *motore di ricerca* è usato per indicare sistemi per il recupero dell'informazione all'interno di basi di dati di documenti HTML generate da uno *spider*[9].

Analogamente ai sistemi per il recupero dell'informazione descritti al capitolo precedente, anche i motori di ricerca sono composti di:

- *spider*: si incarica del reperimento delle informazioni (documenti HTML) da indicizzare, recuperandole direttamente dalla rete; in genere si dice che lo *spider* di un motore di ricerca "visita un sito", intendendo con ciò il recupero di tutte le informazioni contenute in quel particolare sito;

- indicizzatore: indicizza le informazioni rese disponibili dallo *spider*;

- programma di ricerca: estrae dall'indice un elenco di riferimenti ai documenti HTML in base ad una chiave di ricerca espressa dall'utente; ad ogni riferimento viene tipicamente associata una valutazione numerica, che rappresenta una stima del grado di interesse dell'utente nei confronti del documento estratto.

A differenza dei sistemi di recupero dell'informazione tradizionali, i motori di ricerca si propongono di usare come base di dati tutti i documenti di Internet, indistintamente, e di consentire l'accesso al programma di ricerca mediante una connessione client-server: il cliente del servizio invia la chiave di ricerca, il motore (come server) risponde con l'elenco riferimenti trovati.

Le differenze fra questi due tipi di sistemi sono quindi a livello di architettura hardware / software impiegata[2]. I motori di ricerca sono infatti applicativi estremamente complessi soprattutto per le straordinarie condizioni in cui debbono lavorare:

- devono essere in grado di gestire enormi moli di dati; ad esempio, AltaVista (vedere oltre il paragrafo 2.1.2) ha una base di dati di approssimativamente 50 GByte;

- i tempi di risposta devono essere molto brevi (dell'ordine di grandezza del secondo), comunque trascurabili rispetto ai tempi necessari al trasferimento dei risultati attraverso la rete;

- devono poter sopportare un numero molto elevato di richieste di connessione contemporaneamente.

I motori di ricerca devono affrontare inoltre il problema dell'obsolescenza dei dati nell'indice.
I documenti sulla rete, infatti, sono soggetti a qualunque tipo di cambiamento (a totale discrezione di chi li ha pubblicati), mentre la completa indicizzazione del contenuto di tutti i documenti presenti in rete è completata mediamente in tre mesi[9]: quindi l'indice non potrà mai contenere dati completamente aggiornati.

La decisione di quali siti visitare è presa dall'indicizzatore, in base a criteri sia probabilistici che casuali (in pratica, tenderanno ad essere indicizzati più frequentemente i siti che contengono pagine che risultino modificate ad ogni successiva visita[9]).
L'indicizzatore istruisce lo *spider*, fornendogli l'elenco dei siti da visitare. Ogni motore offre la possibilità di integrare questo elenco con siti indicati direttamente dall'utente.
In media, lo *spider* di un motore di ricerca recupera un milione di pagine al giorno.

È possibile suddividere i motori di ricerca in due categorie in base al loro modo di funzionamento: indicizzatori e catalogatori.

Gli indicizzatori creano l'indice basandosi prevalentemente sul contenuto dei siti visitati dal proprio *spider*. Il loro scopo principale è indicizzare quanti più documenti possibile, ed estrarne i riferimenti con le tecniche più rapide e sofisticate. Questo li porta ad adottare archivi enormi per la memorizzazione delle informazioni raccolte.

I catalogatori creano l'indice basandosi prevalentemente sui siti degli utenti che richiedono di essere catalogati. La loro caratteristica principale è operare una categorizzazione delle informazioni raccolte, costruendo una struttura gerarchica (detta catalogo o direttorio) per raggruppare i riferimenti ai siti che trattano argomenti affini. Questo modo di operare offre una visione panoramica di tutto quello che è disponibile in Internet (a meno dei problemi di obsolescenza discussi in precedenza) riguardante un determinato argomento.

La divisione fra catalogatori ed indicizzatori non è comunque netta: esistono infatti catalogatori che dispongono di indici molto vasti, ed indicizzatori che mantengono collezioni di riferimenti a siti ordinate per argomenti.

2.1 Elenco dei più noti motori di ricerca

Vengono ora brevemente descritti i più noti motori di ricerca, mantenendo la suddivisione in catalogatori ed indicizzatori.

2.1.1 Catalogatori

- Yahoo![3]. È il motore che meglio di ogni altro realizza il concetto di categorizzazione, consentendo anche la navigazione all'interno del catalogo: per ogni argomento è mantenuta una corrispondente collezione di sottoargomenti, siti e documenti.
  Molte pagine Web fanno riferimento a Yahoo! per fornire una visione ampia e completa di quanto è disponibile in Internet su un certo argomento; anche molti programmi, per consentire il reperimento di altri con caratteristiche simili, fanno riferimento a Yahoo!;

- Excite[4]. È in grado di usare una ricerca mista: per parole chiave e per argomento. Una volta presentata la lista dei documenti trovati, si può specificare una ricerca per similitudine con uno di questi.

- Lycos[5]. È il più antico dei motori di ricerca. Ultimamente ha aggiornato la propria tecnologia, migliorando anche l'interfaccia grafica.

- WebCrawler[6]. Ha la particolarità di consentire espressioni in "linguaggio naturale", nel senso che è in grado autonomamente di eliminare le cosiddette "stop words".

2.1.2 Indicizzatori

- AltaVista[8]. È probabilmente il motore che dispone dell'indice più vasto.

- HotBot[10]. Consente di recuperare informazioni navigando attraverso una raccolta di riferimenti suddivisi per argomento.

Esistono innumerevoli altri motori di ricerca che non presentano caratteristiche e funzionalità particolari, per i quali valgono quindi le considerazioni generali viste in precedenza.

Alcuni motori hanno versioni dislocate in diversi paesi del mondo, in modo da permettere un accesso molto più rapido alle informazioni e rispettare la "netiquette[1]"; tra questi si ricordano AltaVista, Yahoo!, Excite, Infoseek.

2.1.3 Misti

Sono strumenti di ricerca che funzionano come sistema di interfacciamento verso uno o più veri motori di ricerca:

- ricevono in ingresso le parole chiave con cui estrarre documenti da parte dell'utente,
- sottopongono questa richiesta ai veri motori di ricerca (a più motori contemporaneamente),
- rielaborano i risultati ottenuti in modo tale da presentare in uscita un elenco consistente di riferimenti (eliminazione dei riferimenti doppi e riordinamento dei risultati in base alla valutazione che ogni motore ne ha dato).

Questi strumenti servono sostanzialmente a ridurre i problemi di obsolescenza dei dati negli indici, in quanto uniscono le informazioni presenti in più motori distinti, sfruttando perciò il meglio di ognuno di essi.

Il più noto di questi è, probabilmente, MetaCrawler[11].

2.2 Espressività della chiave di ricerca

Tutti i motori offrono il medesimo grado di espressività della chiave di ricerca, pur se con sintassi leggermente diverse. Per comprendere quali possibilità sono offerte, si prende ad esempio quella di AltaVista.

Come già descritto al paragrafo 1.3, esistono fondamentalmente due tipi di ricerca:

- ricerca semplice, in cui sono estratti i riferimenti ai documenti che contengono almeno una delle parole digitate come chiave di ricerca,
- ricerca complessa, in cui i riferimenti sono estratti adottando come chiave di ricerca una espressione logica (che dispone dunque degli operatori `and`, `or`, `(` e `)`) munita degli ulteriori operatori `near` (impone che

[1]Si intende con ciò la convenzione secondo la quale ci si deve impegnare ad occupare le risorse della rete il meno possibile, cercando cioè di collegarsi con siti che si trovino geograficamente vicini alla propria postazione di lavoro.

le parole si trovino vicine nel testo) e `adj` (impone l'adiacenza fra le parole).

In entrambi i tipi di ricerca sono utilizzabili altri operatori, che hanno lo scopo di limitare il campo a cui la ricerca va estesa:

- `host:<nome>` restringe la ricerca ai soli documenti che risiedono sulla macchina (*host*) specificata;
- `title:<parole chiave>` impone che la ricerca per parole chiave sia limitata al solo campo `<title>...</title>` dei documenti HTML;
- `url:<parole chiave>` restringe la ricerca ai soli documenti che contengono le parole chiave specificate nella propria URL (Uniform Resource Locator, il nome univoco associato ad un qualunque file accessibile in rete);

È anche sempre utilizzabile la ricerca di parole indipendentemente dalla desinenza:

- `radice*`: estrae tutti i riferimenti ai documenti che contengono una parola che inizia per `radice`.

Capitolo 3

Spider

Uno *spider* è un programma che automaticamente attraversa la struttura ipertestuale del World Wide Web, recuperando un documento e ricorsivamente quelli che quest'ultimo riferisce[7].
Recuperare un documento significa copiarne il contenuto da un sito remoto al disco rigido della macchina su cui lo *spider* è in esecuzione.

Gli *spider* a volte sono detti *robot*, in quanto possono funzionare anche solo come semplici esecutori degli ordini impartiti da altri programmi: se è necessaria la presenza di un particolare file, si ordina al robot di recuperarlo.

Sono programmi algoritmicamente piuttosto semplici: non è quindi possibile darne una valutazione comparativa dal punto di vista delle prestazioni offerte, le quali sono dominate dai tempi di risposta del collegamento fisico con il sito remoto da visitare.

Al giorno d'oggi esistono innumerevoli *spider*[15], disponibili in versione commerciale o gratuitamente in rete, dotati delle più disparate caratteristiche.

Essi sono una componente fondamentale nell'architettura di un motore di ricerca e più in generale di un sistema per il recupero di informazioni non locali, nella misura in cui rendono disponibile all'indicizzatore i dati da elaborare.

Vengono ora esaminati nel dettaglio alcuni *spider*, che rappresentano lo stato dell'arte di quanto disponibile in rete attualmente.

3.1 Prodotti esaminati

3.1.1 GNU Wget

Wget è uno *spider* scritto da H.Niksic, rilasciato come componente del sistema operativo Linux. È utilizzabile secondo i termini della licenza d'uso genereale GNU.

Wget è uno *spider* altamente funzionale.
Il suo funzionamento è controllabile mediante un opportuno file di configurazione o mediante corrispondenti parametri sulla linea di comando con cui è eseguito:

- è possibile indicare più siti, che verranno visitati in sequenza;
- ad ogni sito è associabile la coppia di valori *userid:password*[1] nel caso in cui sia protetto contro accessi non autorizzati;
- è possibile decidere se il recupero dei documenti debba essere limitato al primo trovato, o se invece vanno recuperati anche quelli ad esso correlati da riferimenti ipertestuali.

Wget riproduce sul disco rigido la stessa struttura di direttori in cui i documenti visitati si trovano sul sito di provenienza, includendo anche il nome del server corrispondente.

Wget si trova sotto forma di codice binario (eseguibile) per ambienti Unix (Solaris e Linux).

3.1.2 Marauder

Marauder[2] è uno *spider* altamente configurabile. Una sessione di lavoro inizia specificando al programma una pagina HTML di partenza; a questo punto è possibile:

- recuperare solo quella pagina;
- recuperare quella pagina e tutte quelle correlate ad essa da riferimenti ipertestuali;

[1]

[2] ftp.uni-marburg.de/mirror/simtel.net/win95/inet/mdr14.zip
In realtà, tutte le considerazioni fatte su **Marauder** riguardano una *evaluation version*, non la versione commerciale.

- esplorare il sito, cioè generare un elenco delle pagine visitate con i relativi attributi di dimensione e data di ultima modifica senza recuperarne fisicamente il contenuto.

Le informazioni così ottenute dalla rete vengono riorganizzate sulla macchina locale in maniera tale da rispecchiare fedelmente la loro disposizione sulla macchina di provenienza: in pratica viene ricostruita sul disco rigido locale la gerarchia dei direttori in cui le pagine si trovano nel sito remoto, includendo anche il nome del server.

Nel caso sia necessario operare con siti che necessitino dell'uso di *userid:password* per l'accesso, questi valori sono specificabili in fase di configurazione.

Marauder è uno *spider* rilasciato in versione commerciale sotto forma di eseguibile per ambienti Windows 95/NT; è l'unico *spider* esaminato che interagisce con l'utente usando l'interfaccia grafica di Windows.

3.1.3 Acme WebCopy

Acme è una libreria di classi Java (vedere il capitolo 4) che fornisce gli strumenti per un rapido sviluppo di programmi di *spider* e di elaborazione (*parsing*) di documenti HTML. È resa pubblica sul sito:

`http://www.acme.com`

Insieme alla libreria viene fornita l'applicazione **WebCopy**, che ne esemplifica l'uso: è la realizzazione di uno *spider* dalle caratteristiche piuttosto povere, in quanto l'unico compito che è in grado di svolgere è il recupero dei documenti da un sito: non è in grado, ad esempio, di fornire *userid:password* ai siti che ne facciano richiesta, e nemmeno può produrre un elenco dei documenti visitati senza recuperarne il contenuto.

Alla limitazione delle funzionalità offerte si contrappone la caratteristica fondamentale di questo programma, che lo differenzia nettamente dagli altri esaminati in questo capitolo: è scritto in Java ed è rilasciato sotto forma di file sorgenti. Questo significa che può funzionare ugualmente bene sotto diverse piattaforme (Linux, Windows, Solaris, Macintosh, NeXT, OS/2) ed è possibile ampliarne a piacimento le caratteristiche offerte.

3.2 Robots exclusion

In conclusione di un capitolo che tratta dell'argomento degli *spider*, è importante affrontare il tema della cosiddetta "robots exclusion" (esclusione dei robots).

Nel 1993 e nel 1994 si sono verificati parecchi spiacevoli incidenti causati da *spider* che hanno recuperato pagine Web da particolari siti: ad esempio, alcuni *spider* hanno recuperato lo stesso documento più volte, entrando in un loop infinito, altri hanno provato a recuperare programmi CGI che hanno scatenato effetti collaterali indesiderati, altri hanno rallentato le operazioni dei server con richieste troppo ripetitive, ecc...

Questi fatti hanno evidenziato la necessità di stabilire un meccanismo che permetta ai server WWW di indicare agli *spider* quali parti del sito non devono essere accedute.

Il metodo usato per escludere uno o più *spider* dall'accedere a determinate zone di un sito è creare un file che specifichi la politica di accesso per gli *spider* di quel particolare server. Tale file deve essere accessibile via HTTP alla URL

```
/robots.txt
```

Senza entrare nel merito della sintassi del file, diciamo che in `robots.txt` vengono sostanzialmente elencati gli *spider* indesiderati e le zone del sito a cui non è consentito l'accesso.

È stato adottato questo approccio in quanto è estremamente semplice da realizzare da parte del server, e consente allo *spider* di ottenere le informazioni sulla politica di accesso recuperando un solo documento.

In conclusione, è importante sottolineare come questo metodo non sia affatto una reale protezione contro accessi indesiderati da parte degli *spider*: spetta infatti allo *spider* stesso recuperare `robots.txt` e controllare se gli è consentito l'accesso. Non è formalmente vietato scrivere *spider* che *ignorino* il contenuto di `robots.txt`: essi semplicemente non adotteranno questo standard. Tutti gli strumenti esaminati in questo capitolo, d'altro canto, adottano lo standard della "Robots exclusion".

Capitolo 4

Java

L'applicazione da realizzare è un sistema per il recupero delle informazioni, utilizzabile da parte degli utenti mediante l'uso di un *browser*.

Il linguaggio scelto per svilupparla è Java. Questa scelta è motivata dalle seguenti argomentazioni:

- un'applicazione scritta in Java gode delle medesime proprietà che contraddistinguono questo linguaggio: come verrà spiegato più in dettaglio nel prosieguo del capitolo, Java è un linguaggio realmente portabile e moderno;

- l'applicazione da sviluppare basa il suo funzionamento fondamentalmente sull'interazione con la rete e Java è il linguaggio che più e meglio di ogni altro è in grado di integrare all'interno di un programma la gestione delle primitive di rete[1];

- Java ha tali e tanti elementi innovativi, nel panorama odierno dei linguaggi di programmazione, da essere oggetto di numerosi studi e ricerche da parte di università e case produttrici di programmi. L'importanza di Java sarà ancora maggiore se nei prossimi anni si affermeranno i network computer[2], dotati di interprete Java in hardware;

- qualora si prevedesse di estendere l'applicazione con filtri (interfacce) di ingresso ed uscita particolarmente sofisticati dal punto di vista del-

[1] In realtà, anche il C++ è dotato di librerie per la gestione delle primitive di rete; esse risultano però essere non portabili (`libg++` in ambiente Unix, `winsock` in ambiente Windows), ed il loro utilizzo renderebbe quindi l'applicazione vincolata ad una particolare piattaforma.

[2] Computer non dotati di memoria di massa, estremamente robusti e poco costosi, che basano il loro funzionamento sull'interazione con la rete, prelevando e scambiando con essa le informazioni.

l'interazione con l'utente, gli strumenti descritti ed adottati (vedere il capitolo 5) non sarebbero più soddisfacenti e si renderebbe necessario l'uso di Applet Java. Il programma presentato in questa tesi può essere usato indifferentemente come programma CGI (se compilato in codice eseguibile), come Servlet (si veda oltre il paragrafo 4.5) o come Applet: si vedano in proposito il capitolo 8 ed il paragrafo 7.6.

Alla luce di queste considerazioni, segue una descrizione delle caratteristiche del linguaggio Java.

4.1 Introduzione a Java

Java[22][23] è un linguaggio *general purpose*, di alto livello, imperativo e fortemente tipizzato, al pari del C/C++[3] e del Pascal; alcune caratteristiche peculiari di tale linguaggio sono le seguenti:

- Java è un linguaggio di programmazione *orientato ad oggetti*; tre sono i concetti fondamentali introdotti da questa filosofia di programmazione:
  - Incapsulamento: le procedure e le funzioni non sono più definibili in maniera disgiunta dai dati su cui devono operare, cioè i dati del programma incapsulano al loro interno le operazioni per la loro manipolazione. Questo tipo di dato si chiama *classe*; una sua istanza si chiama *oggetto*; le operazioni (procedure e funzioni) definite nella classe si chiamano *metodi*; le operazioni ed i dati definiti all'interno delle classi si chiamano *caratteristiche*.
  - Ereditarietà: una classe può ereditare parte delle proprie caratteristiche da altre, dette *genitori*. Questo tipo di legame può essere visto sia come *specializzazione*, nel senso che le classi eredi definiscono in modo più specifico le caratteristiche dei genitori, sia come *estensione*, nel senso che le classi eredi estendono le possibilità dei genitori.
  - Polimorfismo: consente ad uno stesso metodo di avere realizzazioni differenti in classi differenti; il collegamento fra il nome del metodo chiamato e la sua effettiva implementazione avviene in fase di esecuzione del programma (*binding dinamico*).

  A differenza del C++, che mantiene una forte compatibilità col linguaggio C (quasi tutti i programmi corretti in C lo sono anche in C++),

[3] In realtà il linguaggio C è *debolmente tipizzato*.

Java non ha un modello di riferimento preciso, non è legato a nessun altro linguaggio preesistente. Questa assenza di compatibilità col passato ha permesso di realizzare un approccio orientato ad oggetti reale ed efficace.

- Java è un linguaggio di programmazione *robusto*, nel senso che agevola la scrittura di codice che tende ad impedire un blocco del sistema (evento dannoso) anche in condizioni di funzionamento non previste. La robustezza deriva dal fatto che Java è, in effetti, un linguaggio di più alto livello rispetto ai cosiddetti linguaggi di terza generazione (C, Pascal, ...).

- Java è un linguaggio altamente *portabile*: questo significa poter eseguire gli stessi programmi su piattaforme diverse ottenendo i medesimi risultati. L'approccio adottato per ottenere la portabilità è particolare:

  - i compilatori Java leggono file sorgenti e producono codice eseguibile (*bytecode*) per una macchina virtuale, che fisicamente non esiste (Java Virtual Machine, in breve JVM);
  - la portabilità si ottiene realizzando un interprete di bytecode sulle diverse piattaforme: Linux, Solaris, Windows, NeXT, MacIntosh, OS/2.

  Java è dunque fondamentalmente un linguaggio interpretato; generalmente i programmi scritti in linguaggi interpretati hanno un tempo di esecuzione sensibilmente superiore rispetto a quello ottenibile da un programma analogo compilato in linguaggio macchina. Proprio per questo motivo, gli sforzi maggiori nel progetto dei sistemi Java vengono profusi nel tentativo di mantenere alte prestazioni dal punto di vista della velocità di esecuzione dei programmi, anche a discapito della funzionalità del linguaggio stesso (si vedano in proposito le note riguardanti la libreria di classi nel paragrafo 4.1.2).

4.1.1 Elementi innovativi del linguaggio

"Raccolta degli scarti" (*garbage collection*) automatica

L'uso della memoria è trasparente ai programmatori; essi possono unicamente deciderne l'allocazione mediante l'istanziazione di nuovi oggetti; sarà poi il sistema a determinarne il rilascio a sua totale discrezione.

Il processo di garbage collection avviene tramite il "recupero" di un oggetto. Recuperare un oggetto significa rendere la memoria allocata per esso

di nuovo disponibile al programma. Un oggetto è recuperabile quando non è più accessibile per il programma, quando cioè non è più referenziato da alcun altro oggetto accessibile (sono accessibili gli oggetti a cui puntano le variabili del programma).

Come descritto in [20], esistono fondamentalmente due tipi di algoritmi per realizzare una raccolta automatica degli scarti:

- conteggio dei riferimenti: all'interno di ogni oggetto è aggiunto un campo che mantiene il numero degli oggetti che lo riferiscono; quando questo diviene nullo, l'oggetto è recuperabile.
  Questa tecnica ha grossi svantaggi:
  - spreca tempo: ogni operazione su un riferimento ad un oggetto comporta calcoli e confronti;
  - spreca spazio: è necessario prevedere all'interno di ogni oggetto il campo di conteggio;
  - non è generale: non è in grado, ad esempio, di recuperare strutture cicliche (strutture all'interno delle quali gli oggetti si riferiscono l'un l'altro, ma nessuno di essi è referenziato da alcun oggetto accessibile).

- "raccolta degli scarti al volo" (*garbage collection on-the-fly*): è la tecnica più generale di raccolta degli scarti; si tratta di eseguire in tempi opportuni l'algoritmo di recupero degli oggetti non più accessibili.
  Algoritmi per la raccolta degli scarti possono essere trovati in letteratura in [29][28]. Essi sono generalmente organizzati in due fasi: una fase di marcamento, che attraversa la parte attiva della struttura, contrassegnando come "vivi" gli oggetti incontrati; una fase di pulizia, che attraversa sequenzialmente tutta la struttura, mettendo gli oggetti non marcati in una lista di elementi recuperati.
  Tradizionalmente, i raccoglitori degli scarti vengono attivati su richiesta, quando la memoria disponibile è troppo poca o l'esecuzione del programma è sospesa in attesa di un ingresso da parte dell'utente.
  Il raccoglitore degli scarti dell'ambiente Java può invece sfruttarne il *multithreading*: esso può cioè essere eseguito in un processo distinto, contemporaneamente al programma principale, senza la necessità di essere esplicitamente invocato.

Sono pochi i linguaggi a disporre di un raccoglitore degli scarti al volo; tra questi si ricordano Eiffel[20] e Lisp[27]. L'uso di un raccoglitore automatico degli scarti evita malfunzionamenti dei programmi causati da un maldestro

uso della memoria, evenienza piuttosto frequente e molto dannosa con i tradizionali linguaggi di programmazione. Ha anche il notevole vantaggio di liberare i programmatori dall'assillante problema di una corretta gestione della memoria, i quali potranno così concentrarsi sul funzionamento effettivo del programma.

Gestione delle eccezioni

Col termine "eccezione" si intende, nei sistemi di elaborazione, un evento anomalo o speciale che viene gestito in modo particolare dal sistema medesimo. Nel contesto di molti linguaggi evoluti, come Java, un'eccezione [20] è uno speciale segnale che può essere sollevato da una particolare istruzione e gestito da un'altra, eventualmente in una parte molto diversa del programma. Quando è sollevata un'eccezione, il controllo passa al suo gestore.
Questa facilitazione può semplificare la struttura dei programmi, poichè rende possibile disaccoppiare gli algoritmi per i casi normali dalla gestione dei casi particolari.

Questa caratteristica non è innovativa di per se stessa, in quanto è già presente ad esempio in Ada, Eiffel ed in alcune versioni di C++; è altresì innovativo sia il modo in cui si possono definire e generare eccezioni (in Java le eccezioni sono anch'esse oggetti, eredi della classe `Throwable`), sia l'aver previsto un controllo a livello di compilazione (con l'ausilio della parola chiave `throws`) sui metodi che generano eccezioni.

Multithreading

Introduce primitive di programmazione concorrente all'interno di programmi scritti in Java, consentendo al programmatore di suddividere l'applicazione in più processi (*thread*) che possono essere eseguiti concorrentemente (cioè virtualmente contemporaneamente).
Si noti come lo stesso "raccoglitore degli scarti" sia un *thread* permanentemente in esecuzione, a bassa priorità.

4.1.2 La libreria di classi

Una libreria è un insieme di routines per gestire una particolare risorsa o per manipolare un particolare tipo di dato. Le librerie per gestire le risorse fondamentali della macchina (memoria, dischi) e i tipi fondamentali (interi, stringhe) sono fornite insieme al linguaggio.

Esse sono una componente fondamentale di qualunque linguaggio di programmazione; nel caso dei linguaggi orientati ad oggetti, le librerie (chiamate in questo caso librerie di classi o *class librearies*) assumono ancor maggiore importanza, in quanto devono essere non solo complete ed efficienti, ma anche facilmente estendibili mediante ereditarietà.

Java offre una ricca collezione di librerie, per la gestione di:

- oggetti tradizionalmente presenti in ogni linguaggio: numeri interi, reali, caratteri, stringhe, array;
- strutture dati più complesse: vettori, pile, tabelle hash;
- risorse del disco: file, direttori;
- risorse di rete: astrazione del concetto di URL e di Socket.
- oggetti dell'interfaccia grafica: etichette, campi testo, bottoni, strumenti per disegnare. Questa componente della libreria di classi ha subito profondi cambiamenti tra la versione 1.0.2 e la 1.1.x di Java, causando qualche problema dal punto di vista dello sviluppo delle Applet (vedere il paragrafo 4.3.2).

Come già in precedenza accennato, uno degli obiettivi del progetto Java è mantenere basso il tempo di esecuzione dei programmi pur fornendo un linguaggio interpretato. Questo ha infuenzato il modo in cui le librerie sono state sviluppate; in particolare molte classi fondamentali sono state dichiarate `final`, cioè non estendibili: nessun'altra classe potrà mai ereditare le loro caratteristiche.
Con questa scelta a livello di architettura della libreria di classi si sono scelte le alte prestazioni a discapito della funzionalità. Infatti:

- il fatto di non essere estendibili è assolutamente dannoso dal punto di vista della funzionalità, in quanto spesso costringe i programmatori a scrivere codice poco elegante;
- al contempo, il fatto che una classe non possa avere figli consente al compilatore di produrre codice dall'esecuzione particolarmente più veloce per diversi motivi:
  - una caratteristica fondamentale dei linguaggi orientati ad oggetti è il binding dinamico; esso però tende a rallentare l'esecuzione del programma, in quanto il collegamento fra il nome del metodo chiamato e la sua effettiva realizzazione va calcolato a tempo di esecuzione;

– dichiarando una classe `final`, i suoi metodi non sono ridefinibili e su nessuno di essi è applicabile il binding dinamico; questo fa sì che il tempo necessario per la loro chiamata sia minimo, poichè non coinvolge alcun calcolo.

Se i metodi delle classi che gestiscono i tipi di dati fondamentali (ad esempio stringhe e file) non fossero definiti `final`, l'esecuzione complessiva del programma subirebbe un notevole rallentamento, in quanto il binding dinamico su di essi sarebbe estremamente frequente.

4.2 Java nel mondo *object-oriented*

4.2.1 Java rispetto al C++

Il C++ è un'estensione del linguaggio C che consente la dichiarazione e la manipolazione di classi ed oggetti. Il suo grande vantaggio è al contempo anche il suo massimo limite: è compatibile col C, nel senso che ogni programma ANSI C[25] corretto è anche un programma C++ corretto. Il fatto di avere una organizzazione del programma del tutto simile a quella di un linguaggio assolutamente non orientato ad oggetti come il C, porta il C++ ad essere il linguaggio meno orientato ad oggetti fra quelli che si definiscono tali (per rendersi conto di quanto sia complesso usarlo per produrre codice con caratteristiche orientate ad oggetti si consulti ad esempio [26]); è quindi interessante compararlo con Java.

Java utilizza notazioni sintattiche quasi identiche a quelle del C++ (consentendone quindi un rapidissimo apprendimento da parte di chi già conosca il C/C++), ma ne stravolge completamente l'organizzazione di programma:

- non esistono variabili nè funzioni con visibilità globale: l'unico campo di visibilità è la classe;
- non esistono riferimenti in memoria (puntatori) slegati dai tipi: in Java gli unici riferimenti possibili sono ad oggetti;
- esistono controlli a tempo di esecuzione sulla correttezza dei *type casting*, cioè sulla conversione forzata dei tipi;
- non esistono gli *header* file: gli header file sono associati ad un modulo di programma e ne contengono tutte le dichiarazioni di funzioni, tipi e costanti rese visibili (*esportate*) verso gli altri moduli. La necessità di usare questi file costringe i programmatori C a lunghe riscritture del codice (in quanto la dichiarazione delle funzioni deve apparire nel

modulo ed anche nell'header file), con conseguente possibilità di errori: in Java il luogo in cui trovare le informazioni sulla dichiarazione dei metodi è nella loro stessa definizione.

4.2.2 Java rispetto ad Eiffel

Eiffel è un linguaggio orientato ad oggetti, imperativo, assertivo, nato condividendo parte della sintassi di Ada; è il linguaggio che più di ogni altro dà realizzazione alla filosofia della programmazione ad oggetti[4], ed è per questo l'altro termine di paragone per Java.

In entrambi i linguaggi:

- non esistono funzioni o variabili globali;
- l'esecuzione del programma può iniziare da un qualunque oggetto: non esiste dunque il concetto di programma principale;
- i metodi hanno due modi per terminare: normalmente ritornando un valore, oppure generando un'eccezione;
- non esistono header file: la consistenza sui tipi degli argomenti formali è verificata direttamente sulla definizione del metodo, non su una sua dichiarazione prototipale;

Eiffel è migliore di Java nella misura in cui:

- è un linguaggio assertivo, nel senso che sono previsti a livello di sintassi dei costrutti che consentono la definizione di asserzioni (cioè condizioni che devono essere verificate quando il flusso di programma passa per un determinato punto): precondizioni, invarianti, postcondizioni, all'interno dei metodi e delle classi;
- le classi eredi possono ridefinire ogni caratteristica implementata dalle classi genitori: non esiste cioè l'equivalente della dichiarazione Java di un metodo *private*. Si veda in proposito il problema sorto nel paragrafo 6.2.5;

[4]In realtà, esiste un altro linguaggio, pietra miliare nella storia dei linguaggi orientati ad oggetti, che è doveroso citare: Smalltalk.
Esistono fondamentalmente due categorie in cui suddividere i linguaggi orientati ad oggetti:

- quelli in cui le classi sono distinte dagli oggetti, come Eiffel e Java;
- quelli in cui le classi sono oggetti, come Smalltalk.

Smalltalk è in assoluto il linguaggio più orientato ad oggetti che esista; Eiffel più di ogni altro dà realizzazione alla filosofia della programmazione ad oggetti, limitatamente ai linguaggi della prima categoria.

- ha ereditarietà multipla;
- esiste la dichiarazione per associazione, con la quale si dichiara il tipo di una variabile come essere uguale al tipo di un'altra.

4.3 Applet

Un'Applet[22][23] è un programma scritto in Java e compilato, che può essere richiamato da una pagina HTML ed eseguito in una JVM realizzata all'interno di un *browser*[5]. L'obiettivo di questo tipo di applicazioni è incapsulare interattività ed intelligenza nelle pagine Web.

4.3.1 Limiti di HTML

HTML[21][19] è un linguaggio di definizione **logica** della struttura di un ipertesto.

Un ipertesto[21] è composto da un testo ed una serie di riferimenti che rimandano ad altre informazioni (come avviene nei riferimenti bibliografici). Nel Web tali riferimenti ipertestuali sono realizzati mediante l'ausilio del concetto di URL (che identifica univocamente una risorsa, un documento).
La documentazione ipertestuale non ha una maniera univoca di essere letta (a differenza dei testi normali, che generalmente vengono letti sequenzialmente), in quanto il lettore può accedere alla medesima informazione attraverso diversi riferimenti.

Un aspetto fondamentale di HTML è il fatto che tale linguaggio non determina univocamente come l'ipertesto dovrà apparire agli occhi di chi lo leggerà (in questo si differenzia ad esempio da PostScript, che è un linguaggio di **descrizione** della pagina), serve invece solo ad individuare le varie parti di cui è composto (ed è in questo molto simile al linguaggio LaTeX): il modo effettivo in cui l'ipertesto sarà visualizzato è in realtà deciso dal *browser* che interpreta ed esegue i comandi HTML.

HTML è fondamentalmente un "linguaggio di scripting"[22]. Tali linguaggi mettono a disposizione dei programmatori un insieme predefinito di oggetti, comandi; la programmazione consiste sostanzialmente nel loro utilizzo e nella loro interazione per ottenere il risultato desiderato. Sono strumenti

[5]In realtà, questo è il modo di funzionare dei *browser* "tradizionali", come Netscape ed Internet Explorer.
Esiste un *browser* scritto completamente in Java (HotJava, storicamente la prima applicazione di questo linguaggio[22]), il quale esegue le Applet sulla JVM su cui esso stesso è in esecuzione.

estremamente potenti, poichè la programmazione è molto semplice e permette di ottenere buoni risultati in tempi brevi; diventano immediatamente inutilizzabili nel caso si debba svolgere qualche compito non espressamente previsto all'interno degli oggetti e dei comandi predefiniti.

Perciò HTML è un linguaggio estremamente potente per quanto riguarda le possibilità di rappresentazione logica di un ipertesto, ma assolutamente carente dal punto di vista dell'interattività con l'utente che lo visualizza attraverso il proprio *browser*.

Esistono linguaggi, come ad esempio JavaScript[19], che estendono HTML dal punto di vista delle capacità di calcolo e di controllo di flusso. Tuttavia, essi rimangono pur sempre linguaggi di scripting, che si limitano ad aggiungere comandi ad HTML senza stravolgerne la natura di funzionamento. I comandi di tali linguaggi sono anch'essi interpretati ed eseguiti dal *browser*; questo determina problemi di portabilità poichè *browser* diversi spesso adottano specifiche diverse.

4.3.2 HTML ed Applet

Le estensioni di HTML sono molto comode ed efficaci da usare, ma non sono soddisfacenti nei casi in cui sia necessaria la massima flessibilità di programmazione, ad esempio nel caso si utilizzasse il *browser* come strumento di interfacciamento (*front-end*) verso programmi gestionali, oppure per eseguire particolari elaborazioni grafiche animate.

Le Applet sono la giusta soluzione in questi casi, poichè sono in grado di fare all'interno di un *browser* su una macchina client tutto quanto si potrebbe fare in una normale finestra dell'interfaccia grafica sul proprio computer. Questo rende possibile sviluppare applicazioni per fornire servizi attraverso la rete in cui l'interfaccia sia intelligente: ciò significa svolgere sulla macchina del client parte dell'elaborazione che altrimenti sarebbe svolta per intero dal server; ad esso rimane ora sostanzialmente l'unico compito di recuperare e trasferire i dati verso il client.

Come già detto, tecnicamente un'Applet è un programma scritto in Java ed eseguito da una JVM realizzata all'interno del *browser*; l'applicazione viene caricata sulla macchina del cliente dalla pagina HTML che sta visualizzando, al pari di un'immagine o di un suono, e qui mandata in esecuzione. Questo comporta due ordini di problemi.

Sicurezza

È una caratteristica irrinunciabile del progetto Java. Infatti, per rendere

accettabile l'idea di caricare un programma dalla rete e mandarlo in esecuzione sul proprio computer, il linguaggio con cui esso è scritto deve assicurare caratteristiche di sicurezza straordinarie. In questo contesto trovano la loro massima giustificazione i concetti di astrazione ed indipendenza dall'architettura hardware presenti in Java, senza i quali queste caratteristiche non sarebbero ottenibili[22].
Ad esempio, per ragioni concernenti la sicurezza, alle Applet è impedito scrivere o leggere arbitrariamente documenti dalla macchina su cui sono in esecuzione.

Portabilità

Ogni *browser* "tradizionale" realizza una propria JVM al suo interno, quindi la totale portabilità delle Applet è solo teorica e non è vero in assoluto che funzionano allo stesso modo su qualunque *browser*. Inoltre, sussistono problemi per quanto riguarda le differenze fra versioni diverse.
La situazione attuale, ad esempio, è la seguente: è disponibile la versione 1.1.5 dell'ambiente di sviluppo JDK (vedere il paragrafo 4.4.1), mentre i browser ancora implementano JVM basate sulla versione 1.0.2. Tra queste due versioni esistono grossi cambiamenti, soprattutto dal punto di vista del modello dell'interfaccia grafica, anche se la 1.1 mantiene (temporaneamente) compatibilità con la 1.0.2.
I programmatori sono così costretti a sviluppare Applet compatibili con 1.0.2 (altrimenti non funzionerebbero sui browser tradizionali), che presto diverranno obsolete e andranno adeguate con la 1.1.

4.4 Ambienti per lo sviluppo di applicazioni Java

4.4.1 Sun Java Development Kit 1.1.3

Il Java Development Kit (JDK[17]) è il pacchetto per lo sviluppo di applicazioni Java, siano esse programmi o Applet, prodotto dalla stessa casa (Sun Microsytems) che ha inventato il linguaggio e ne detiene le specifiche. La stessa Sun si impegna a distribuirne realizzazioni per varie piattaforme hardware: Solaris, Linux, Windows 95/NT, MacIntosh, NeXT, OS/2. Il JDK è gratuitamente ottenibile da Internet.
Segue una breve descrizione dei principali programmi che compongono la distribuzione.

- **javac**: è il compilatore; traduce un programma sorgente (`.java`) in bytecode (`.class`).

- **java**: è la Java Virtual Machine; è in grado di interpretare il bytecode ed eseguirlo su una particolare macchina.

- **javadoc**: è uno strumento innovativo[6]; serve a produrre la documentazione tecnica su un programma estraendo le informazioni direttamente dal sorgente corrispondente; queste informazioni vanno inserite sotto forma di particolari commenti. La documentazione prodotta viene organizzata in documenti HTML, contenenti l'elenco dei metodi e delle variabili definiti all'interno delle varie classi.
  L'unica documentazione completa della libreria di classi di cui è fornito Java è stata prodotta da javadoc.

  Javadoc risolve il fondamentale problema di mantenere aggiornata (congruente) la documentazione di un software rispetto alla sua implementazione; questo aspetto della programmazione è particolarmente importante quando lo sviluppo avviene modularmente:

  - i clienti di un modulo lo usano basandosi sulla sua documentazione;
  - se tale documentazione non è prodotta congiuntamente allo sviluppo del modulo sarà sempre obsoleta, e non ne descriverà la reale implementazione;
  - javadoc fa coincidere la fase di sviluppo del codice con quella di produzione della documentazione, risolvendo il problema dell'incongruenza.

  Per un esempio di qual è il risultato ottenibile da javadoc si consulti l'allegato con la documentazione prodotta su parte del programma oggetto di questa tesi.

- **appletviewer**: serve ad eseguire e testare le Applet anche senza l'ausilio di un *browser*.

- **jre**: è una versione ridotta della Java Virtual Machine; serve sostanzialmente a produrre programmi Java *standalone*, cioè caricabili senza necessità di richiamare esplicitamente l'interprete.

[6] In realtà, uno strumento del tutto analogo è già previsto ed usato nei sistemi Eiffel: si chiama **short**.

4.4.2 Ambienti integrati di sviluppo

Sun Microsystems produce anche un ambiente di programmazione integrato di nome Java WorkShop; è sostanzialmente un'interfaccia grafica al JDK, scritta in Java stesso, per semplificarne l'utilizzo ed automatizzare lo sviluppo dei programmi.

Esistono numerosi altri ambienti integrati di sviluppo prodotti da diverse case produttrici di software: Symantec Visual Café Pro, SuperCede, Microsoft Visual J++ per piattaforme Windows.

4.5 Java e programmi CGI

Le risorse offerte dall'ambiente Java consentono di realizzare diversi tipi di programmi.

È infatti possibile realizzare programmi *standalone*, cioè programmi in esecuzione sul proprio computer esplicitamente richiamati dall'utente, semplicemente invocando la JVM (o utilizzando **jre**, la differenza è minima) prima della loro esecuzione.

Come già spiegato, sono realizzabili anche Applet, eseguite sulla JVM presente all'interno dei browser.

Lo sviluppo altresì di programmi CGI è fonte di parecchi problemi, che verranno ora esaminati nel dettaglio.

I programmi CGI vengono richiamati in genere da una form HTML, la quale tipicamente mette a loro disposizione una serie di valori testuali digitati dall'utente; questi ultimi vengono processati dal programma CGI, il quale genera come risultato un nuovo testo HTML.

Il primo problema all'utilizzo di un linguaggio interpretato come Java per realizzare tali programmi risiede nel fatto che nell'attributo `action` della form HTML non è possibile specificare un programma con dei parametri: se così non fosse, sarebbe possibile utilizzare un programma Java come CGI mediante la seguente sintassi:

```
<html>
  ...
  <form method=... action="java CgiScript">
    ...
  </form>
  ...
</html>
```

(dove `CgiScript` rappresenta l'applicazione da eseguire).

Una soluzione a questo problema consiste nello sviluppare un piccolo programma, con un linguaggio compilato (ad esempio il C), il cui unico compito sia quello di invocare l'interprete Java con gli opportuni argomenti. Ad esempio, il codice della form si modificherebbe nel modo seguente

```
<html>
  ...
  <form method=... action="boot.exe">
    ...
  </form>
  ...
</html>
```

dove `boot.exe` è un programma eseguibile il cui codice è simile al seguente (scritto in C):

```
main() {
  /* eventuale lettura dei parametri inviati dalla form...  */
  system("java CgiScript");
}
```

Questo metodo funziona correttamente su sistemi Unix (Linux, Solaris, ...), nei quali il programma eseguibile `boot.exe` può addirittura essere scritto anche nel linguaggio della shell del sistema operativo. D'altro canto, questo metodo non funziona su sistemi Windows: all'atto della chiamata `system`, l'output prodotto dal processo figlio (`java CgiScript`) è perduto.

Una soluzione drastica a questo problema consiste nell'utilizzo di compilatori Java in grado di produrre codice *nativo* per Windows (o meglio, produrre codice per processori Intel x86), cioè direttamente eseguibile sulla macchina fisica senza la necessità di richiamare la JVM. Gli unici compilatori dotati di questa capacità sono Symantec Visual Café Pro e Asymetrix SuperCede. Adottando questa tecnica, la form HTML verrebbe coì riscritta:

```
<html>
  ...
  <form method=... action="CgiScript.exe">
    ...
  </form>
  ...
</html>
```

Questo tipo di soluzione ha il grosso svantaggio di essere estremamente onerosa dal punto di vista delle risorse da impiegate per l'esecuzione del programma: esso, infatti, necessita della presenza di un grande numero di DLL[7], associate al compilatore, che devono essere caricate e scaricate dalla memoria del computer ogni volta che il programma CGI viene invocato, determinando un forte decadimento complessivo delle prestazioni della macchina e dei tempi di risposta del CGI[8].

Una soluzione molto suggestiva ma poco pratica consiste nell'utilizzo di un programma di gestione del Web (Web Server) scritto completamente in Java. Tale soluzione è suggestiva in quanto i programmi CGI verrebbero eseguiti sulla medesima JVM su cui è in esecuzione il Web Server. È altresì poco pratica in quanto non è generalmente accettabile il vincolo di avere un Web Server scritto in Java: in genere si preferisce, per motivi di efficenza e affidabilità, utilizzare i prodotti più tradizionali per questo tipo di applicazioni (Apache per sistemi Unix, IIS per sistemi Windows). Il Web Server scritto in Java prodotto dalla Sun è disponibile all'indirizzo:

```
http://jeeves.javasoft.com/products/java-server/index.html
```

La soluzione che risolve radicalmente il problema dell'utilizzo di Java per sviluppare programmi CGI è senza dubbio anche la più elegante e la più efficace: l'uso delle **Servlet**.
Le Servlet sono programmi scritti in Java ed eseguiti come fossero programmi CGI quando l'utente accede ad una particolare URL. Il programma che intercetta l'accesso dell'utente a questa particolare URL va installato come *filtro* del Web Server. In definitiva, il funzionamento è di questo tipo:

- se l'utente accede alla URL corrispondente alle Servlet, generalmente del tipo

  ```
  http://www.server.it/servlet/CgiScript
  ```

  allora il programma installato come filtro assume il controllo del sistema: il programma Java corrispondente alla richiesta dell'utente (nell'esempio `CgiScript`) viene eseguito, invocando la corrispondente JVM,

[7]Una DLL (Dynamic Link Library) è un insieme di programmi (una libreria), che sono resi disponibili ad una applicazione in fase di esecuzione. Normalmente, quando un programma necessita di una funzione definita in una particolare DLL deve preventivamente caricare la DLL medesima in memoria, mediante l'uso di un metodo messo a disposizione dai vari linguaggi; ad esempio, in Java tale metodo si chiama `loadLibrary(String name)`.

[8]Le considerazioni esposte in questa tesi a riguardo dei compilatori Java per produrre codice nativo sono basate sull'utilizzo di SuperCede ver.1.2.

ed i risultati riportati esattamente come per il funzionamento dei tradizionali programmi CGI; si noti come in questo caso il programma di gestione del Web Server non venga interpellato;

- se l'utente accede ad una qualunque altra URL, il controllo è ceduto normalmente al programma di gestione del Web Server.

In questo caso, la form HTML risulterebbe di questo tipo:

```
<html>
  ...
  <form method=... action="http://www.server.it/servlet/CgiScript">
    ...
  </form>
  ...
</html>
```

Adottando questo approccio, sono parecchie le ottimizzazioni dal punto di vista delle risorse impiegate che è possibile ottenere:

- la JVM è caricata una sola volta in memoria ed è unica per tutti i programmi CGI (Servlet) da eseguire; la gestione delle richieste multiple da parte di più utenti della medesima Servlet si basa sul multithreading dell'ambiente Java;

- è previsto un metodo `init()`, in cui è possibile concentrare tutte le operazioni (tipicamente la configurazione del programma) che devono essere eseguite una sola volta e richiedono un tempo particolarmente lungo per il loro completamento; ad una successiva invocazione della medesima Servlet, l'inizializzazione non sarà più eseguita di nuovo, garantendo così un ben più rapido funzionamento;

- le Servlet sono programmi che rimangono in memoria dopo la prima esecuzione; ad una successiva invocazione non solo non eseguiranno più la fase di inizializzazione, ma non sarà nemmeno più necessario caricarle dal disco rigido;

Un vasto elenco di prodotti che consentono la gestione delle Servlet è consultabile all'indirizzo:

`http://jserv.javasoft.com/products/java-server/servlets/environments.html`

Il prodotto utilizzato per l'utilizzo delle Servlet sviluppate in questa tesi è **JRun 2.0** prodotto da Live Software, gratuitamente disponibile nel sito:

`http://www.livesoftware.com/products/jrun/`

4.6 Esempi

4.6.1 Programmazione Java

Una *classe* è dichiarata mediante l'uso della parola chiave `class`:

```
class UnaClasse {
  ...
}
```

All'interno della classe possono essere dichiarati metodi e variabili:

```
class UnaClasse {
  protected String unaVariabile;
  public int unMetodo( String unParametro ) {
    ...
  }
}
```

Un *oggetto* può essere creato istanziando una classe:

```
UnaClasse unOggetto;              // dichiarazione
unOggetto = new UnaClasse();      // creazione
```

Su questo oggetto possono essere richiamati i metodi dichiarati nella classe a cui appartiene:

```
int unIntero;
unIntero = unOggetto.unMetodo("una stringa");
```

Come già accennato, è interessante esaminare il modo in cui Java tratta le eccezioni. Un metodo che generi eccezioni deve indicarlo nella sua dichiarazione; il metodo seguente genera eccezioni provocate da malfunzionamenti di periferiche di ingresso / uscita:

```
class UnaAltraClasse {
  public void unMetodoConEccezioni() throws IOException {
    ...
  }
}
```

Un metodo può gestire i casi di eccezione facendo opportuno uso del costrutto sintattico `try..catch`:

```
class UnaClasse {
  public void metodoChiamante() {
    ...
    UnaAltraClasse unAltroOggetto = new UnaAltraClasse();
    try {
      unAltroOggetto.unMetodoConEccezioni();
      ...  // blocco di gestione caso normale
    } catch (IOException e) {
      ...  // blocco di gestione dell'eccezione
    }
  }
}
```

4.6.2 Uso del compilatore

Per compilare un file sorgente di nome `UnSorgente.java` e produrre il corrispondente bytecode `UnSorgente.class` il comando da eseguire è il seguente:

```
javac UnSorgente.java
```

Per eseguire il bytecode precedentemente prodotto bisogna richiamare la Java Virtual Machine:

```
java UnSorgente
```

Parte 2

Capitolo 5

Architettura del sistema e specifiche

Lo scopo principale di questa tesi è stata la definizione e lo sviluppo di un prototipo completo di sistema per il recupero delle informazioni in un preciso contesto applicativo. Infatti, la presente tesi è stata svolta in collaborazione con Talete, società del gruppo Zuffellato, con la quale il Dipartimento di Ingegneria ha una convenzione nell'ambito dell'iniziativa comunitaria "Adapt (priorità Adapt Bis – Building the Information Society)" dal titolo "*Servizi multimediali interattivi per lo sviluppo del commercio internazionale: la PMI entra nella società dell'informazione*".

Scopo del sistema è quello di reperire attraverso la rete informazioni sul commercio internazionale, rendendole fruibili attraverso una interfaccia con caratteristiche "intelligenti" ad utenti inesperti, appartenenti al mondo della Piccola e Media Impresa.

5.1 Scopo del progetto

Tale progetto si fonda sulla necessità che la Piccola e Media Impresa (PMI) approdi all'uso di Internet quale strumento di informazione e comunicazione. La realizzazione del progetto in questione consentirà alle PMI, in generale non fornite dei mezzi per fronteggiare l'aumento della competitività dovuta al processo di globalizzazione dei mercati, di ampliare i propri mercati di riferimento grazie ad un sistema informativo completo, relativamente alle problematiche legate alla compravendita internazionale.

Gli obiettivi principali del progetto sono i seguenti:

- Soddisfare il crescente bisogno delle Piccole Medie Imprese di avere a disposizione direttamente sui propri terminali Internet un sistema

informativo completo in materia fiscale, giuridica, economica, valutaria, doganale e finanziaria.

- Minimizzare i tempi di reperimento delle informazioni: le aziende, grazie all'utilizzo del programma attraverso Internet, potranno accedere direttamente all'insieme di informazioni contenute nella BASE DI DATI del sistema informativo e porre direttamente delle domande ad esperti dei vari settori, i quali elaboreranno una risposta pubblicamente accessibile.

- Ridurre i costi per l'ottenimento delle informazioni: il sistema informativo sostituirà la necessità di impiegare risorse umane esterne al fine di reperire dati.

La realizzazione della banca dati quale guida esaustiva alle problematiche legate all'import-export, risponde all'esigenza delle Piccole e Medie imprese, destinatarie del servizio, di avere a disposizione le informazioni per l'esaurimento delle varie fasi di sviluppo di una transazione commerciale, dalla ricerca del contraente, alla stipula del contratto internazionale, dalla scelta del mezzo ideale di pagamento e riscossione, alla delineazione dei termini per la consegna, ecc...

Il progetto è innovativo sotto diversi punti di vista:

- dal punto di vista delle risorse metodologiche, comporta l'utilizzo di nuove tecnologie quali Internet e Java e di teniche di Intelligenza Artificiale;

- dal punto di vista dei contenuti, comporta la realizzazione di un sito multimediale in grado di raccogliere e rendere disponibili varie sorgenti di informazioni mediante un unico sistema informativo: testi di documentazione presenti in rete, informazioni contenute in banche dati, interazione con gli esperti dei vari settori;

- dal punto di vista della struttura, integra e fonde conoscenza umana ed artificiale.

Tale progetto prevede meccanismi suscettibili di favorire la riproducibilità e la trasferibilità delle prassi migliori: tali meccanismi fanno parte della natura intrinseca del progetto in quanto la realizzazione del sistema informativo in questione presuppone la realizzazione di un modello (prototipo) valido per tutti i paesi dell'Unione Europea. L'adattamento del servizio alle esigenze ed alle particolarità degli altri paesi membri sarà un processo favorito dalla esaustiva impostazione della banca dati.

5.2 Specifiche di progetto

Oggetto di questa tesi è lo sviluppo del sistema di recupero dell'informazione, ossia della componente più importante del sistema informativo che rappresenta il servizio complessivo descritto nel progetto comunitario.

Tale sistema di recupero informazioni opera su una base di dati composta da una serie di documenti presenti in rete. L'operazione di ricerca e selezione delle fonti di informazione (siti contenenti *documenti*) in rete è estremamente critica, data la vastità delle possibili sorgenti; deve inoltre essere condotta in modo efficiente poichè influenza direttamente l'utilità del servizio finale. Per questo motivo, l'indicazione di quali siti e documenti esplorare viene fornita da esperti nel settore del commercio internazionale, che suggeriscono al sistema dove trovare i dati. Tale insieme di dati utili distribuiti sulla rete viene definito come BASE DI DATI (vedere figura 5.1).

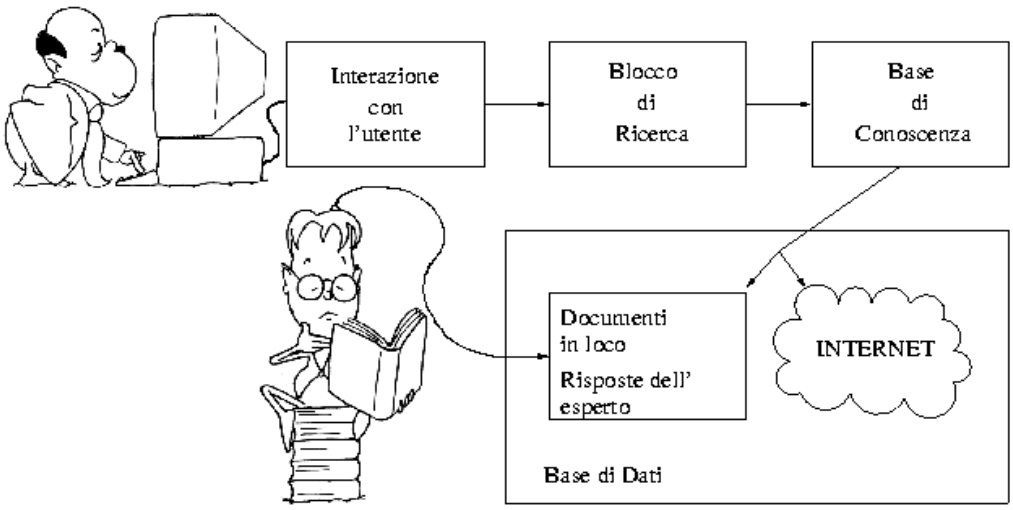

Figura 5.1: Funzionamento di principio

Il sistema per il recupero delle informazioni deve consentire all'utente di fare una interrgoazione mediante chiavi di ricerca e deve estrarre una serie di riferimenti ai documenti ad essa corrispondenti che si trovano nella BASE DI DATI.
L'estrazione è resa possibile e rapida dalla preliminare opera di indicizzazione

della BASE DI DATI: a tal proposito, nel progetto si indica con il termine BASE DI CONOSCENZA l'insieme degli indici prodotti da questa indicizzazione condotta in modo automatico dal sistema.

Rispetto ad un sistema per il recupero delle informazioni tradizionale, il programma sviluppato in questa tesi si differenzia in almeno due aspetti fondamentali.

1. **Categorizzazione**
   L'insieme dei documenti che compongono la BASE DI DATI è stato suddiviso in categorie. Tale categorizzazione (suggerita dagli esperti dei vari settori che raccolgono le fonti di informazione) è la caratteristica che contraddistingue questo sistema per il recupero delle informazioni: potendo gestire il concetto di categoria, infatti, esso può fornire all'utente una visione più immediata e precisa degli argomenti disponibili; lo scopo è di migliorare l'efficacia e l'efficienza del sistema guidando l'utente a fare ricerche più mirate.

2. **Gestione dell'esperto in linea**
   Con l'espressione "esperto in linea" si intende un esperto umano che interagisce con il sistema artificiale nel seguente modo: l'esperto genera documenti che il sistema artificiale automaticamente integra nella BASE DI DATI. I documenti generati sono le risposte alle domande poste dagli utilizzatori del sistema, nel caso in cui essi non abbiano avuto sufficienti informazioni dalla ricerca all'interno della BASE DI DATI.

L'interazione fra utente ed esperto in linea è molto differente da quella fra utente e BASE DI CONOSCENZA:

- la domanda posta all'esperto può essere del tutto generale, e grammaticalmente comunque complessa;

- la risposta dell'esperto è fornita con un periodo di ritardo comunque molto superiore rispetto ai tempi di risposta del programma di estrazione dei riferimenti (in genere un esperto umano risponde entro 24–48 ore, mentre il programma risponde immediatamente, con tempi che dipendono fondamentalmente dalla velocità del collegamento Internet, comunque dell'ordine delle decine di secondi);

- la risposta fornita dall'esperto in linea non è inviata direttamente all'utente che ha posto la domanda: l'insieme delle risposte viene infatti integrato nella BASE DI DATI, in una particolare categoria della BASE DI CONOSCENZA. L'utente può quindi consultare le risposte sia

direttamente (accedendo al direttorio in cui si trovano le risposte, che possiamo nominare "Case Studied Resolved", o "direttorio CSR" in breve), sia mediante interrogazione del sistema per il recupero delle informazioni.

Dal punto di vista realizzativo, il sistema può dunque essere scomposto nei seguenti blocchi funzionali (vedere figura 5.2), per ognuno dei quali seguirà una dettagliata descrizione delle specifiche progettuali da rispettare:

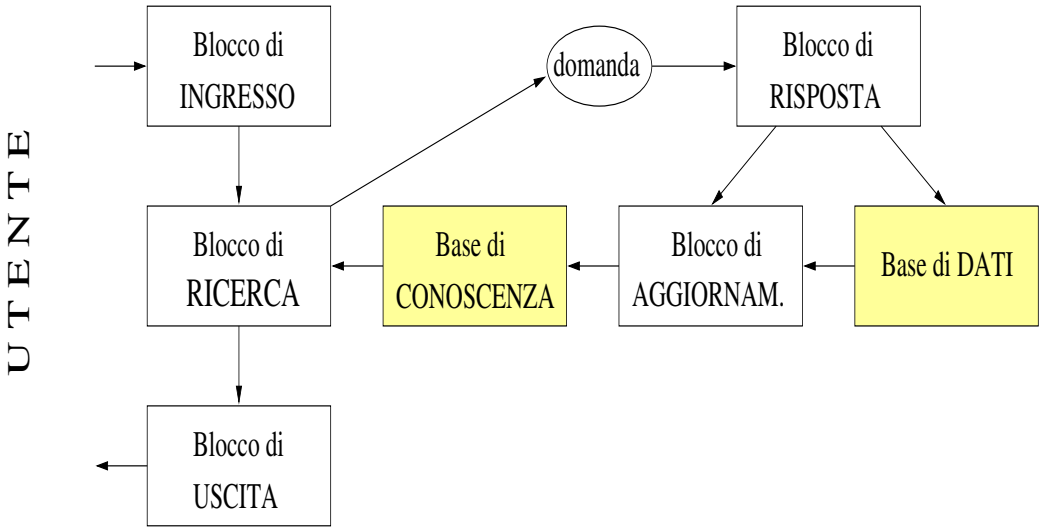

Figura 5.2: Schema a blocchi funzionali

- **BASE DI DATI**: è l'elenco delle sorgenti di informazione reperibili su Internet e sul sito di Talete che andranno indicizzate;
- **BASE DI CONOSCENZA**: è l'indice strutturato con categorie della base di dati;
- **BLOCCO DI AGGIORNAMENTO**: consente la manutenzione delle informazioni presenti nella BASE DI CONOSCENZA;
- **BLOCCO DI INGRESSO**: consente all'utente di esprimere la chiave di ricerca o la domanda da sottoporre all'esperto in linea;
- **BLOCCO DI RICERCA**: recupera i riferimenti ai documenti che corrispondono alla chiave di ricerca dalla BASE DI CONOSCENZA;
- **BLOCCO DI USCITA**: presenta all'utente i risultati del BLOCCO DI RICERCA;

- **BLOCCO DI RISPOSTA**: consente all'esperto di rispondere ad una **domanda** posta da un utente.

Per il server che renderà disponibile agli utenti il sistema informativo, è stato deciso da Talete l'impiego di un sistema operativo Windows NT Server 4.0.

5.2.1 BASE DI DATI

Come già spiegato, la BASE DI DATI è l'insieme dei documenti HTML su cui verterà l'applicazione del sistema di recupero dell'informazione.

I documenti si trovano in numerosi siti Internet di pubblico dominio; sono perciò soggetti a cambiamenti in qualunque momento, a totale discrezione di chi li ha pubblicati.

Un documento può entrare a far parte della BASE DI DATI in tre maniere distinte:

- per essere stato esplicitamente dichiarato dagli esperti che operano la raccolta delle sorgenti di informazione;
- per essere correlato ipertestualmente ad un documento esplicitamente dichiarato nella BASE DI DATI;
- per essere un documento prodotto dall'esperto in linea: tali documenti devono essere integrati in maniera automatica nella BASE DI DATI e si trovano fisicamente nella memoria di massa del server di Talete nel direttorio CSR (vedere il prossimo paragrafo).

5.2.2 BASE DI CONOSCENZA

La BASE DI CONOSCENZA è l'indice del sistema di recupero delle informazioni, costruito indicizzando il contenuto di ogni documento che compone la BASE DI DATI.

Essa ha una struttura gerarchica: è composta da diverse categorie di argomenti e sottoargomenti; ogni documento della BASE DI DATI si trova in una o più categorie della BASE DI CONOSCENZA.

Analogamente a quanto discusso a proposito dei motori di ricerca al capitolo 2, anche la BASE DI CONOSCENZA soffre di problemi di obsolescenza, a causa del fatto che i documenti indicizzati sono soggetti a cambiamenti da parte di chi li ha pubblicati in rete.
Questo determina la necessità di sviluppare anche un modulo con il compito di mantenere la coerenza fra le informazioni nella BASE DI CONOSCENZA

e quelle nella BASE DI DATI (BLOCCO DI AGGIORNAMENTO, vedere il paragrafo 5.2.3).

La struttura gerarchica stessa è soggetta a variazioni, che hanno comunque periodicità molto inferiore rispetto a quelle dei suoi contenuti.

La struttura della BASE DI CONOSCENZA è completamente descritta in un unico file. Esso deve:

- fornire i nomi di ogni categoria; ognuna ha un nome esterno (*etichetta*, presentata all'utente) ed un nome interno (che identifica univocamente la categoria per il programma);
- specificare la suddivisione delle categorie in argomenti e sottoargomenti; per tale rappresentazione si è scelta una struttura di dati *ad albero*;
- elencare, per ogni categoria, l'insieme dei documenti di cui è composta; per ogni documento è possibile (ma non strettamente necessario) specificare:
  - una etichetta, che è il nome con cui quel documento sarà visualizzato all'utente;
  - l'eventuale necessità, da parte del BLOCCO DI AGGIORNAMENTO degli indici, di recuperare anche i documenti correlati da riferimenti ipertestuali a quello in esame.

All'interno di questo file è necessario prevedere anche una categoria particolare, contenente l'indice delle risposte dell'esperto in linea (che possiamo brevemente chiamare categoria CSR). Essa è del tutto analoga alle altre, può trovarsi in una qualunque posizione dell'albero delle categorie; si distingue unicamente per il fatto di avere un nome interno particolare: questo consente al programma di individuarla e gestirla opportunamente.
A questo punto, è opportuno sottolineare la differenza fra due diversi concetti: categoria CSR e direttorio CSR.

- il direttorio CSR contiene una serie di documenti HTML, ognuno dei quali è formato sostanzialmente dal testo della domanda posta dall'utente e dalla corrispondente risposta fornita dall'esperto. Il direttorio CSR fa parte della BASE DI DATI;
- la categoria CSR individua anch'essa un direttorio sul disco rigido del server, il quale contiene però semplicemente l'indice (prodotto dal programma di indicizzazione) del direttorio CSR; essa consente all'utente di ricercare per parole chiave anche all'interno del direttorio CSR. La categoria CSR fa parte della BASE DI CONOSCENZA.

Il file di descrizione della struttura della BASE DI CONOSCENZA è stato chiamato `z.cat`; si veda il paragrafo A.2 per esaminarne il contenuto e la sintassi effettiva.

5.2.3 BLOCCO DI AGGIORNAMENTO

Il BLOCCO DI AGGIORNAMENTO è il programma che ha il compito di mantenere la corrispondenza fra le informazioni contenute nella BASE DI DATI e la BASE DI CONOSCENZA. Essendo un programma di manutenzione del sistema, la sua esecuzione dovrebbe essere permessa esclusivamente all'amministratore del sistema medesimo.

Concettualmente, il BLOCCO DI AGGIORNAMENTO riceve in ingresso il file `z.cat` e produce in uscita una BASE DI CONOSCENZA aggiornata.

Più in dettaglio, per ogni categoria della BASE DI CONOSCENZA, esso deve conoscere:

- la posizione nel disco rigido in cui si trova l'indice dei documenti associati alla categoria; questa informazione viene costruita basandosi sui nomi interni dati alle varie categorie;

- la posizione sulla rete in cui si trovano i documenti da indicizzare per la categoria; questa informazione si trova esplicitamente nel file `z.cat`.

Con queste informazioni, per ogni categoria il programma di aggiornamento procede a generare un testo (*rapporto*) che, per ogni documento HTML associato alla categoria in esame nella BASE DI DATI, elenchi nome, dimensione e data di ultima modifica.
È in questa operazione che si rende necessaria la presenza di uno *spider* che sappia generare un elenco dei documenti visitati (indicandone appunto solo nome, dimensione, data di ultimo aggiornamento), senza recuperarne il contenuto.

Comparando il rapporto testè generato con quello corrispondente stilato durante la precedente esecuzione dell'aggiornamento, il programma è in grado di capire se le informazioni presenti nella BASE DI CONOSCENZA sono obsolete rispetto a quelle presenti nella BASE DI DATI.
Per eseguire questa comparazione si può fare semplicemente uso del programma di utilità distribuito col sistema operativo Dos/Windows `fc.exe`; ugualmente bene funziona `cmp.exe` prodotto da GNU.

Se i due rapporti coincidono, le informazioni sono aggiornate: nessuna azione va ulteriormente intrapresa, ed il programma prosegue con l'elaborazione della categoria successiva.

Se i due rapporti differiscono, è necessario provvedere all'aggiornamento dell'indice associato alla categoria in esame.
Questo significa innanzitutto procedere al trasferimento sulla macchina locale del contenuto di tutti i documenti che compongono la categoria. Chiaramente, per espletare questo compito è necessario disporre di un programma di *spider*. Esso deve però soddisfare specifiche molto particolari:

- l'elenco dei documenti da recuperare viene fornito mediante un file, situato nel direttorio corrispondente alla categoria;
- i documenti scaricati vanno copiati in un particolare direttorio sul disco rigido, in modo tale che il programma che effettivamente crea l'indice possa trovarli;
- è fondamentale che per ogni documento recuperato venga riprodotta sul disco rigido una struttura di direttori esattamente corrispondente a quella in cui essi si trovano sul sito remoto, includendo alla radice di questa struttura lo stesso nome del server da cui provengono.

In particolare, questa terza specifica consente di tener traccia all'interno dell'indice del nome completo (la URL) del documento scaricato. In effetti, il risultato che il sistema di recupero di informazioni dovrà produrre come risposta ad una chiave di ricerca è la URL, ed è per questo motivo che essa va inserita nell'indice assieme al documento.

Dopo aver ottenuto sul proprio disco rigido l'insieme delle informazioni che compongono la categoria, l'operazione di aggiornamento ha termine con la loro indicizzazione e conseguente aggiornamento dell'indice all'interno della BASE DI CONOSCENZA.
Questo è un compito svolto dall'indicizzatore. Esso riceve in ingresso il direttorio in cui si trovano le informazioni da indicizzare e produce in uscita l'indice corrispondente, il quale va posizionato nel direttorio corrispondente alla categoria.
L'unica specifica particolare che questo applicativo deve soddisfare è poter essere eseguito e configurato da un altro programma, non da un operatore umano: l'indicizzatore viene infatti richiamato una volta per ogni categoria da aggiornare ed ogni volta deve elaborare e produrre archivi differenti.

L'operazione di aggiornamento di una categoria ha termine con l'eliminazione dal disco locale dei documenti scaricati durante la fase di spidering: il riferimento ad essi sarà d'ora in poi accessibile accedendo all'indice. Viene ovviamente aggiornato anche il rapporto corrispondente alla categoria, rimuovendo quello obsoleto.

L'operazione di sostituzione dell'indice obsoleto con quello aggiornato solleva problemi di condivisione di risorse. Per una approfondita discussione di

questi problemi e della loro soluzione dal punto di vista realizzativo si rimanda alla sezione 7.1.2.

Il BLOCCO DI AGGIORNAMENTO deve prevedere anche la possibilità di aggiornare esclusivamente il direttorio CSR e produrre l'indice nella categoria CSR.
Quando viene impartito questo particolare comando (evidentemente dal BLOCCO DI RISPOSTA dell'esperto), il BLOCCO DI AGGIORNAMENTO deve semplicemente richiamare l'indicizzatore, fornendogli i seguenti due parametri:

- il direttorio CSR sul disco rigido (in cui si trovano i documenti con le risposte fornite dall'esperto) da indicizzare; questa informazione è determinata in sede di configurazione del sistema informativo, e dipende fondamentalmente dalla struttura data al sito Internet che offre il servizio;

- il direttorio sul disco rigido corrispondente alla categoria CSR (in cui si trova l'indice delle risposte fornite dall'esperto); questa informazione è ottenibile dal file `z.cat`, tenendo presente che il direttorio di una categoria è associato al suo nome interno, e la categoria delle risposte dell'esperto ha un nome univoco, particolare.

Alla fase di indicizzazione fa seguito anche in questo caso quella di sostituzione dell'indice obsoleto con quello aggiornato, che analogamente a prima fa sorgere problemi di condivisione di risorse.

Dal punto di vista realizzativo, il BLOCCO DI AGGIORNAMENTO è un programma, in esecuzione sul server fornitore del servizio.

5.2.4 BLOCCO DI INGRESSO

Il BLOCCO DI INGRESSO è quella parte del sistema per il recupero delle informazioni che consente all'utente di esprimere una chiave di ricerca.

La chiave di ricerca è composta da due parti:

- La prima parte (opzionale) è l'insieme degli argomenti (ovvero delle categorie) a cui l'utente vuole restringere la ricerca. È opzionale in quanto, se l'utente non specifica nulla, si assume voglia coinvolgere nella ricerca l'intera BASE DI CONOSCENZA.
  Come già accennato in precedenza, il fatto di avere una struttura suddivisa per argomenti e sottoargomenti consente al BLOCCO DI INGRESSO di fornire all'utente gli strumenti per una ricerca più mirata

ed efficace. Egli deve infatti poter selezionare in maniera estremamente semplificata (graficamente sulla struttura ad albero) gli argomenti di suo interesse. Per tale possibilità si prevede quindi un'interfaccia GRAFICA di navigazione e selezione delle categorie sulla struttura ad albero (vedere figura 5.3).
Sarà poi il BLOCCO DI RICERCA a prendersi carico della decisione di quali indici della BASE DI CONOSCENZA interessare nell'effettiva ricerca.

- La seconda parte (necessaria) della chiave di ricerca è un elenco di parole in base alle quali estrarre i riferimenti ai documenti. L'utente gode di una totale libertà nella scelta di codeste parole e può, in effetti, digitare anche una frase in linguaggio naturale.
  L'utente non dovrebbe comunque scrivere frasi complesse nè particolarmente articolate, in quanto non viene effettuata alcuna analisi semantica su di esse. Il BLOCCO DI INGRESSO tenterà comunque di estrarre dalla frase digitata le parole chiave significative per la ricerca, operando due diverse elaborazioni:

  - eliminazione delle "stop words", nel tentativo di mantenere solo la parte semanticamente significativa della frase,
  - applicazione di un algoritmo di "stemming"[37] alle parole rimaste, nel tentativo di rendere la ricerca più generale.

Il modo in cui appare l'interfaccia per la lettura della chiave di ricerca è visualizzato nella figura 5.3

Il BLOCCO DI INGRESSO ha anche il compito di consentire all'utente di formulare una domanda da inoltrare verso l'esperto in linea. Una tale domanda si compone di tre parti:

- uno o più argomenti che determinano l'ambito della domanda; la selezione degli argomenti (categorie) avviene in maniera identica a prima, graficamente sull'albero delle BASE DI CONOSCENZA;
- il titolo della domanda, che è il nome con cui l'utente troverà la risposta;
- il testo della domanda, che a differenza di prima non ha alcun vincolo di complessità dal punto di vista dell'articolazione delle frasi, essendo letto da un operatore umano.

Uno degli obiettivi a cui il progetto tenderà in futuro sarà unificare questi due tipi di richiesta di informazioni al sistema informativo: si veda il paragrafo 8.2 per maggiori dettagli.

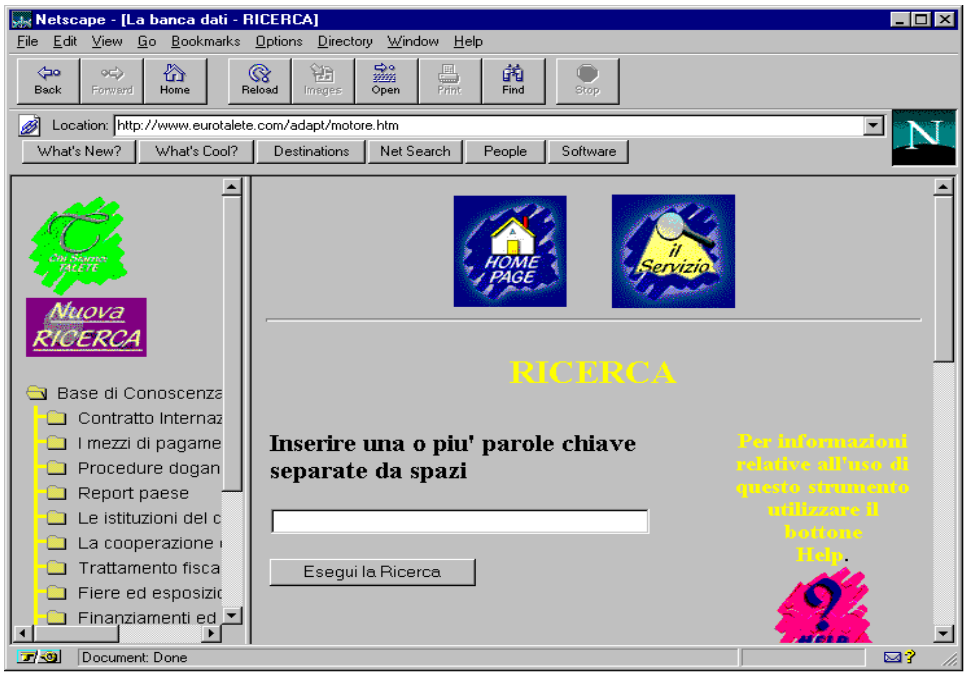

Figura 5.3: BLOCCO DI INGRESSO–Chiave di ricerca

Il modo in cui appare l'interfaccia per la lettura del titolo e del testo della domanda da sottporre all'esperto è rappresentato nella figura 5.4.

Dal punto di vista realizzativo, il BLOCCO DI INGRESSO può essere sviluppato con il linguaggio HTML, con l'ausilio di JavaScript, o anche mediante Applet Java.

5.2.5 BLOCCO DI RICERCA

Il BLOCCO DI RICERCA ha una duplice natura: oltre alle caratteristiche di tradizionale estrattore di riferimenti, esso deve infatti anche poter funzionare come programma che inoltra le domande degli utenti all'esperto.

Estrattore di riferimenti

Nel funzionamento come estrattore di riferimenti, il BLOCCO DI RICERCA riceve dal BLOCCO DI INGRESSO la chiave di ricerca, composta dall'elenco di categorie selezionate e dalle parole chiave per cui estrarre i riferimenti.

L'unica operazione che deve compiere è richiamare il programma che effettivamente realizza l'estrazione dei riferimenti (tipicamente il medesimo che ha eseguito l'indicizzazione), fornendogli i seguenti due parametri:

Figura 5.4: BLOCCO DI INGRESSO–Domanda all'esperto

- la successione degli indici in cui ricercare; i nomi degli indici sono calcolati dal BLOCCO DI RICERCA in base all'elenco delle categorie selezionate dall'utente nel BLOCCO DI INGRESSO: essi corrispondono dunque fondamentalmente alla prima parte della chiave di ricerca;

- l'elenco delle parole chiave per cui ricercare all'interno degli indici.

I risultati prodotti dall'indicizzatore vengono direttamente inviati al BLOCCO DI USCITA, il quale li presenta all'utente.

Il funzionamento come estrattore di riferimenti è indipendente dalla gestione dell'esperto in linea. Come già in precedenza spiegato, l'indice dei documenti con le risposte prodotte dall'esperto corrisponde ad una categoria della BASE DI CONOSCENZA la quale, in quanto categoria, viene gestita dal BLOCCO DI RICERCA esattamente come le altre; semplicemente, i riferimenti estratti da tale categoria sono riferimenti a documenti che fisicamente risiedono sullo stesso server fornitore del servizio.

Gestione dell'esperto

Nel funzionamento come programma che inoltra le domande all'esperto, il BLOCCO DI RICERCA riceve dal BLOCCO DI INGRESSO le tre compo-

nenti della domanda: categorie, titolo e testo.

Esso ha il compito di costruire un documento HTML, contenente il testo della domanda, in un particolare direttorio del disco rigido, preposto alla memorizzazione delle domande poste dagli utenti. Possiamo per brevità denominare tale direttorio "direttorio QUEST". Il documento creato ha come nome il titolo della domanda unito alla data in cui è stata posta.

L'effettivo nome del direttorio QUEST è determinato in fase di configurazione del servizio, e dipende da come è strutturato il sito. L'accesso a tale direttorio non dovrebbe essere consentito agli utenti; solo l'esperto dovrebbe poter collegarsi per rispondere. Per i dettagli su come avviene la risposta si veda il paragrafo 5.2.7.

La creazione del documento con la domanda solleva altri problemi di condivisione di risorse. In particolare, se non sono previsti meccanismi adeguati, il sistema non è in grado di memorizzare correttamente le domande con lo stesso titolo inviate contemporaneamente da due utenti. Per una descrizione della soluzione di questo problema si rimanda al paragrafo 7.1.2.

Dal punto di vista realizzativo, il BLOCCO DI RICERCA deve necessariamente essere un programma CGI. Essi, infatti, sono l'unico tipo di programmi che sono in grado di scambiare informazioni con una macchina *client* attraverso l'uso di un *browser*, producendo risultati basati sull'elaborazione di dati locali al *server*.

5.2.6 BLOCCO DI USCITA

Il BLOCCO DI USCITA ha il compito di presentare all'utente i risultati della ricerca, cioè fondamentalmente l'elenco dei riferimenti ai documenti della BASE DI DATI che corrispondono alla chiave di ricerca da lui espressa.

La visualizzazione dei risultati deve essere coerente con l'interfaccia di ingresso: all'utente viene presentata la porzione della struttura ad albero della BASE DI CONOSCENZA in cui sono contenuti i documenti trovati. In questo modo egli ha una visione immediata dell'ambito a cui un particolare documento fa riferimento ed è in grado di comprendere in qual misura esso può rispondere alla sua richiesta di informazioni.

Dal punto di vista realizzativo, il BLOCCO DI USCITA è analogo a quello di ingresso, quindi scritto in HTML o JavaScript, ed è in realtà il testo prodotto dal programma CGI che realizza il BLOCCO DI RICERCA. Nel caso in cui l'ingresso fosse realizzato mediante Applet, il BLOCCO DI USCITA sarebbe parte dell'Applet stessa.

Il modo in cui vengono rappresentati i risultati della ricerca nell'applicazione sviluppata è vsibile nella figura 5.5

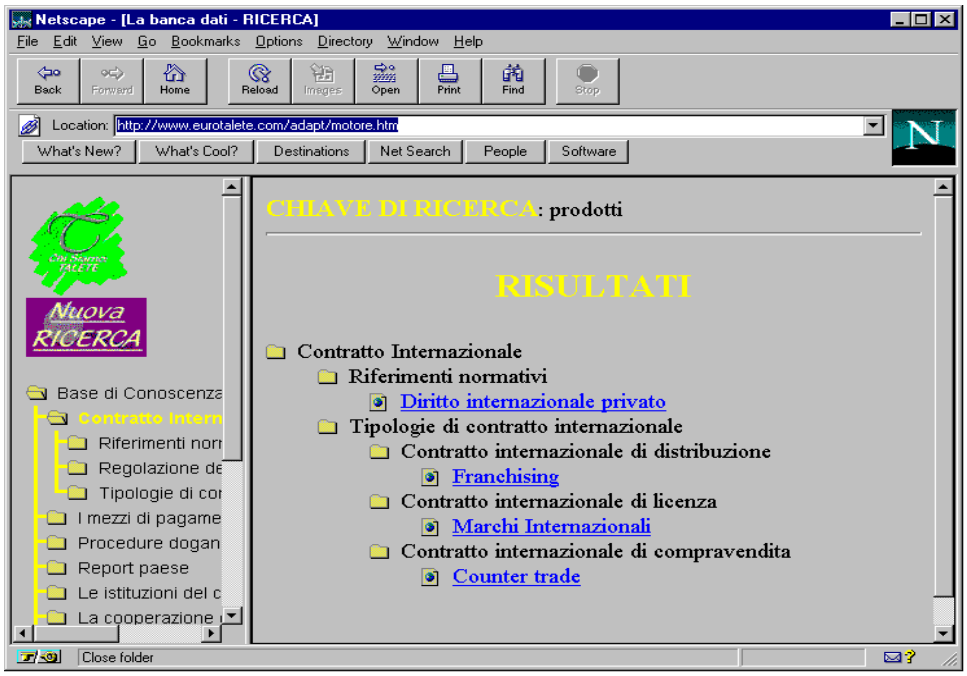

Figura 5.5: BLOCCO DI USCITA

5.2.7 BLOCCO DI RISPOSTA DELL'ESPERTO

Il BLOCCO DI RISPOSTA dell'esperto è un programma che consente all'esperto di rispondere alle domande poste dagli utenti, in maniera assolutamente automatizzata.

Il meccanismo che consente di passare dalla formulazione della domanda alla pubblicazione della risposta è piuttosto complesso, e vale la pena di riesaminare nel dettaglio le varie operazioni compiute dai diversi blocchi funzionali:

- dal BLOCCO DI INGRESSO l'utente seleziona alcune categorie, scrive il titolo ed il testo della domanda;
- queste informazioni vengono raccolte dal BLOCCO DI RICERCA (che in questo caso sarebbe più opportuno chiamare blocco di "domanda"), che crea nel direttorio QUEST (a cui solo l'esperto può avere accesso) un documento HTML dal seguente contenuto:
  - data di invio della domanda;
  - titolo;
  - categorie selezionate;

  - testo della domanda;
  - *form* HTML contenente un campo (un `textarea`, per intenderci) in cui andrà scritta la risposta; l'attributo `action` della presente form fa riferimento al programma che consente la risposta dell'esperto.

- per rispondere alle domande, l'esperto non deve far altro che collegarsi via Internet al sito, entrare nel direttorio QUEST e caricare nel proprio *browser* uno qualunque dei documenti HTML che vede listati (che corrispondono all'elenco delle domande inevase); a questo punto egli dovrà semplicemente scrivere la risposta e sottomettere la form;

- la sottomissione della form richiama il BLOCCO DI RISPOSTA, fornendogli in ingresso tutte le informazioni di cui necessita (data, titolo, categorie, domanda, risposta). Tale blocco dovrà compiere le seguenti operazioni:

  1. creare il documento HTML con il testo della domanda e la relativa risposta all'interno del direttorio CSR;
  2. ricreare, all'interno di questo stesso direttorio, un ulteriore documento HTML contenente l'indice delle risposte fornite; esso serve a rendere più semplice il reperimento, da parte dell'utente, delle risposte alle varie domande; il modo in cui tale indice appare è graficato nella figura 5.6;
  3. comandare al BLOCCO DI AGGIORNAMENTO di riindicizzare il direttorio CSR;
  4. cancellare dal direttorio QUEST il documento a cui è stata appena data risposta;
  5. produrre un testo che assicuri l'esperto che tutte le operazioni sono andate a buon fine.

5.3 Scelta dei componenti

Partendo dalla semplice scomposizione funzionale descritta dalla figura 5.2, vengono ora esaminate diverse soluzioni per quanto riguarda i blocchi di ricerca e di aggiornamento, ottenute componendo opportunamente gli strumenti descritti ai capitoli 1, 2, 3.

Figura 5.6: Indice del direttorio CSR

Nessuno di questi strumenti è in grado da solo di soddisfare le specifiche imposte sul funzionamento complessivo del sistema per il recupero delle informazioni. Un'attività fondamentale di questa tesi è stata perciò, accanto alla progettazione dell'intero sistema, la scelta dei componenti e la realizzazione dei programmi di interfaccia e di cominucazione fra gli stessi.

Gli strumenti software che andranno a costituire il sistema per il recupero delle informazioni dovranno godere delle seguenti due fondamentali caratteristiche:

- configurabilità: essi saranno utilizzati come una singola componente all'interno di una nuova applicazione, quindi in un ambito diverso rispetto a quello in cui tradizionalmente si trovano; è quindi necessario che essi sappiano *adattarsi* alle nuove condizioni di lavoro;

- interfacciabilità: osservando lo schema a blocchi ci si rende conto come l'interazione avvenga fra programmi, coinvolgendo l'utente solo all'inizio ed alla fine dell'elaborazione; è quindi necessario che gli strumenti software utilizzati possano comunicare fra loro direttamente; questo esclude completamente i programmi che adottino un'interfaccia grafica.

5.3.1 Impiego di un motore esterno

La soluzione che comporta il minor impiego di risorse locali dal punto di vista dello spazio occupato per la BASE DI CONOSCENZA è senza dubbio l'impiego di un motore di ricerca remoto (descritti al capitolo 2) come strumento per la memorizzazione dell'indice, il recupero delle informazioni e la ricerca (vedere figura 5.7).

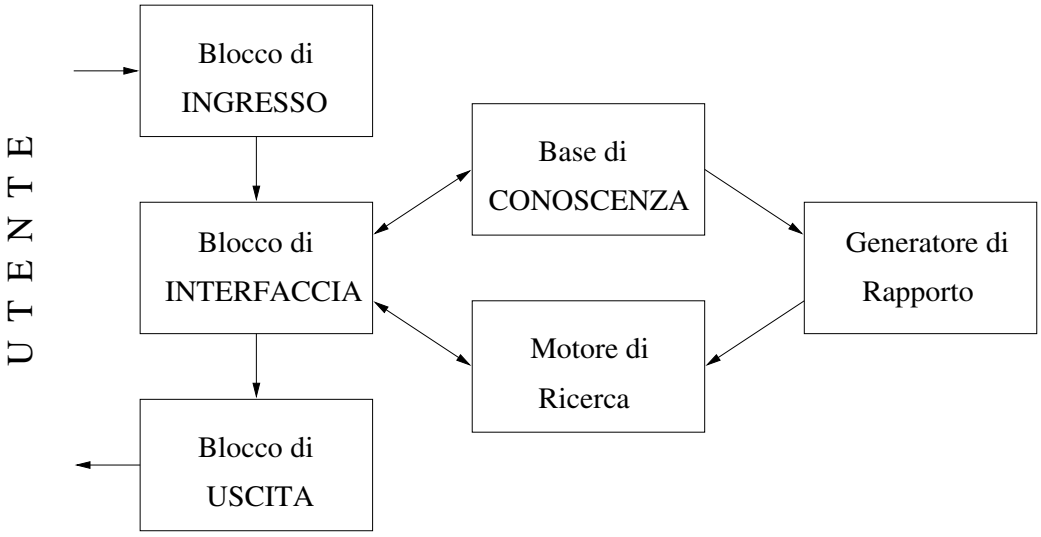

Figura 5.7: Soluzione impiegando un motore di ricerca

Adottando questa soluzione, localmente bisognerà prevedere:

- **Blocco di Interfaccia**: è un opportuno modulo di interfacciamento verso il motore, in grado di tradurre la frase digitata dall'utente in chiavi di ricerca sintatticamente corrette per il particolare motore con cui si opera, inviarle e leggere le risposte corrispondenti. Questo modulo non presenta particolari complessità dal punto di vista algoritmico; ha altresì il problema fondamentale di essere completamente dipendente dal particolare motore di ricerca utilizzato;

- **Generatore di Rapporto**: è un programma di utilità che si incarica di mantenere aggiornate le informazioni nell'indice del motore, indicandogli i particolari siti che si desidera vengano indicizzati. Perciò basta scrivere un modulo che periodicamente controlli la data di ultimo aggiornamento delle informazioni che compongono la BASE DI DATI all'interno dei siti di interesse, imponendo al motore di aggiornare (cioè rivisitare) quelli che risultano cambiati dall'ultima indicizzazione.

Questa possibile soluzione risulta però non soddisfacente a causa dei seguenti problemi:

- in generale, i motori di ricerca non indicizzano completamente un sito che gli sia stato indicato, trascurandone arbitrariamente una parte[9]: il motore di ricerca impone infatti un limite allo spazio occupato da un qualunque sito nel suo indice.
  Un comportamente del genere viola la specifica di progetto che impone di mettere a disposizione degli utenti fruitori del servizio informazioni sempre aggiornate;

- non si può mai essere certi dell'avvenuta indicizzazione, in quanto non si può controllare direttamente il funzionamento dello *spider* associato al motore;

- non è inoltre possibile indicizzare siti che siano protetti contro accessi non autorizzati: non si può specificare ad un motore la chiave d'accesso *userid:password*.

Alla luce di queste considerazioni, si può affermare che l'approccio basato sull'uso di un motore di ricerca remoto è efficace quando vi è la necessità di mantenere indicizzate grandi quantità di dati, contenute massivamente in un numero limitato di siti.

Con riferimento all'applicazione da sviluppare, non è certamente questo il metodo da usare, in quanto:

- la gestione dell'esperto andrebbe trattata come caso particolare, comportando complicazioni dal punto di vista dello sviluppo del programma: non è infatti pensabile far indicizzare il direttorio CSR al motore di ricerca ogni volta che l'esperto formula una risposta;

- le categorie constano principalemente di un lungo elenco di documenti singoli, dislocati in numerosi siti; indicando al motore di indicizzare tali documenti, esso tenderà invece a recuperare l'intero contenuto del sito corrispondente, senza alcuna garanzia di indicizzare l'unico documento che interessava;

- non è possibile mantenere separati gli indici di categorie distinte, ed in generale risulta estremamente complesso sfruttare la categorizzazione operata sulla BASE DI CONOSCENZA. Tale categorizzazione deve essere usata per rendere il servizio offerto maggiormente comodo ed efficace, e non è pensabile adottare una soluzione realizzativa che ne limiti i benefici.

Questo è il motivo fondamentale che impedisce l'uso di un motore di ricerca remoto come strumento per la memorizzazione ed il mantenimento degli indici.

5.3.2 Scelta del sistema di recupero di informazioni

Scartata l'ipotesi di demandare il maggior carico di lavoro ad un motore di ricerca remoto, non rimane che realizzare localmente un sistema di recupero di informazioni non locali, in grado cioè di recuperare autonomamente dalla rete il contenuto della BASE DI DATI, generare la BASE DI CONOSCEZA (gli indici) ed interrogarli; tale sistema sarà composto inglobando gli strumenti descritti ai capitoli 1 e 3.

L'unico strumento che soddsfa alle specifiche di configurabilità e interfacciabilità precedentemente descritte è **SWISH**: è dunque questo il programma usato come indicizzatore ed estrattore di riferimenti.

Gli altri pacchetti esaminati al capitolo 1 non sono altrettanto validi per i seguenti motivi:

- l'uso di **ht://Dig** è reso impossibile dal vincolo di progetto che impone di sviluppare l'applicazione per ambiente Windows NT[1];
- l'uso di **Personal AltaVista** è reso impossibile in quanto l'*indexer* di tale pacchetto viola la condizione di interfacciabilità: esso sfrutta infatti appieno le caratteristiche dell'interfaccia grafica di Windows, rendendo comoda l'interazione con un operatore umano, ma assolutamente impossibile lo scambio di dati con un altro programma; ad esempio, è impossibile costruire un applicativo che configuri l'*indexer* per indicizzare un particolare elenco di files, oppure variare la periodicità con cui l'indice viene ricostruito, e nemmeno è possibile chiudere l'*indexer* quando non serve più.

5.3.3 Scelta dello spider

Il fatto di adottare **SWISH** come strumento per la ricerca e l'indicizzazione dei documenti comporta la necessità di introdurre un altro componente nel sistema: uno *spider* che renda disponibili per l'indicizzazione localmente sul disco rigido le informazioni contenute nella BASE DI DATI.

[1]In realtà, **ht://Dig** è scritto in C++, che è un linguaggio portabile; non sono però portabili le librerie per l'uso delle primitive di rete di cui la componente di spider (htdig) del pacchetto necessita.

Nel progetto in esame serve un componente di spider dalla duplice natura (vedere il paragrafo 5.2.3):

- recupero del contenuto di un testo;
- generazione di una lista dei testi da recuperare (con i relativi attributi di dimensione e data di ultimo aggiornamento).

La soluzione adottata consiste nell'uso di una versione opportunamente modificata di **WebCopy**. Questo programma, infatti, soddisfa le condizioni iniziali imposte:

- è configurabile, nel senso che, essendo rilasciato sotto forma di sorgenti, è completamente modificabile ed estendibile;
- è interfacciabile, in quanto non usa l'interfaccia grafica per interagire con l'utente.

Inoltre, essendo disponibile sotto forma di sorgenti Java, è liberamente modificabile fino a soddisfare qualunque specifica.

Gli altri programmi *spider* esaminati al capitolo 3 non sono altrettando validi in quanto:

- analogamente a quanto detto a proposito di **ht://Dig**, **Wget** non è portabile sotto Windows, in particolare in quanto è disponibile esclusivamente sotto forma di codice binario esegubile per ambienti Unix;
- al pari dell'*indexer* di **Personal AltaVista**, **Marauder** è un programma assolutamente non integrabile in un contesto che non preveda la costante presenza di un operatore umano: il suo comportamento è determinato dall'interfaccia grafica guidata ad eventi di Windows, il che ne preclude la possibilità di utilizzo.

Capitolo 6

Impiego e modifica di programmi preesistenti

La effettiva realizzazione dell'applicazione per il recupero delle informazioni è dunque l'assemblaggio di diversi componenti: alcuni sono strumenti già esistenti, altri vanno sviluppati completamente.

In questo capitolo verranno discusse le modifiche apportate agli strumenti già esistenti per integrarli nell'applicazione per il recupero delle informazioni da sviluppare, secondo le specifiche date al capitolo 5.

6.1 Migrazione di SWISH

Come spiegato al capitolo precedente, **SWISH** è il programma usato per la realizzazione del BLOCCO DI RICERCA e di AGGIORNAMENTO (fare riferimento alla figura 5.2).

La versione originale di **SWISH** funziona su sistemi operativi Unix, quindi è necessario provvedere a migrarlo (*porting*) dall'ambiente Unix sotto l'ambiente Windows, in quanto l'applicazione andrà sviluppata con questo sistema operativo.

SWISH è scritto in C, e fa esclusivamente uso di librerie per la gestione dei file: questo lo rende compilabile anche in altri ambienti, nella fattispecie Windows. Il fatto di poter compilare senza errori sintattici il sorgente di un programma non garantisce che sia privo anche di errori logici di funzionamento. Infatti, per completare la migrazione è necessario modificare le parti di codice che dipendono dalle caratteristiche differenti dei due sistemi operativi.

6.1.1 Adeguamento del sorgente alle specifiche ANSI C

In effetti, **SWISH** è scritto nel C detto di Kernighan e Ritchie[25] e non è direttamente compilabile da una compilatore che adotti le specifiche ANSI. Per la compilazione in ambiente Windows si è utilizzato il compilatore Watcom C/C++ ver. 10.5, che per l'appunto soddisfa alle specifiche ANSI. Il primo sforzo per rendere compilabile l'insieme dei moduli di programma ha dunque comportato l'adeguamento del sorgente alle suddette specifiche:

- riscrittura della dichiarazione e definizione di ogni funzione, comprendendo anche la dichiarazione dei tipi degli argomenti formali. Ad esempio, la funzione dichiarata originalmente

```
void indexadir(dir)
     char *dir;
{
  ...
}
```

  è stata modificata in:

```
void indexadir(char *dir)
{
  ...
}
```

- esplicitazione del prototipo di ogni funzione: le specifiche ANSI prevedono che si disponga della dichiarazione (prototipo) di una funzione prima della sua chiamata. Sostanzialmente, questo meccanismo ha lo scopo di:
  - consentire al compilatore numerosi controlli formali sulla consistenza fra i tipi dei parametri attuali (del chimante) e dei parametri formali (della dichiarazione);
  - esplicitare le relazioni fra moduli: ogni modulo ha un corrispondente header file (vedere il paragrafo 4.2.1) in cui inserisce, tra l'altro, anche tutte le dichiarazioni delle funzioni che definisce. Ogni modulo *cliente* dovrà dunque esplicitamente includere l'*header* file di tutti i moduli di cui fa uso (i *fornitori*).

Questo si traduce in un problema con il codice di **SWISH**, poichè ogni modulo include solo il proprio *header* file: molte funzioni in ogni modulo sono chiamate senza conoscerne il prototipo. Il C Kernighan e Ritchie supplisce alla mancanza del prototipo fornendone uno implicito[25], l'ANSI C termina la compilazione con un errore.

Per semplicità, è stato allora scritto un *header* (`vice.h`) che include l'header di tutti i moduli di programma; si è poi proceduto alla loro modifica, imponendogli anche l'inclusione di `vice.h`. In questo modo, ogni modulo dispone del prototipo di ogni funzione del programma.

Certamente non è una soluzione elegante, ma l'alternativa sarebbe stata entrare nel dettaglio dell'architettura di **SWISH**, studiarne le relazioni fra i moduli ed esplicitarle, determinando per ognuno i fornitori ed includere gli opportuni *header*.

In generale, si è cercato di limitare per quanto possibile modifiche così intrusive nell'architettura del programma, che sarebbero state potenzialmente fonte di numerosi errori.

6.1.2 Gestione dei diversi simboli di EOLN

In ambiente Unix, il simbolo per marcare EOLN (end-of-line, separatore fra le righe in un file di testo) coincide con un carattere singolo: `LF` (Line Feed, codice ASCII 10).
In ambiente Windows, EOLN è la sequenza di caratteri: `CR` (Carriage Return, codice ASCII 13) ed `LF`.

Il linguaggio C ha la particolarità di traslare automaticamente il simbolo `CR LF` in `LF` durante la lettura, e `LF` in `CR LF` in scrittura verso un file che sia stato aperto con l'attributo `t` (`text`, file di testo).

L'indice generato da **SWISH** non è un file di testo, cioè non è composto da soli caratteri alfanumerici. Quindi se avvenisse questa traslazione, il contenuto del file risulterebbe corrotto. Dal punto di vista delle modifiche da apportare al codice, è necessario specificare l'attributo `b` (`binary`) ad ogni apertura del file dell'indice. Ad esempio

```
fopen(index, "w")
```

diventa

```
fopen(index, "wb")
```

Esiste un altro punto nel testo che necessita di modifica a causa del diverso simbolo usato per EOLN: la funzione `parsetitle(char *)` nel modulo

`string.c`.
Essa deve memorizzare nella variabile locale `title` la stringa corrispondente al titolo del documento HTML che si sta indicizzando, il quale è delimitato nel testo dalle espressioni `<title>` fino a `</title>`. Da tale stringa risultante dovranno essere eliminati tutti i caratteri *bianchi* (cioè spazi, tabulazioni orizzontali e verticali, form-feed, line-feed e carriage-return) in testa ed in coda, e non dovrà contenere EOLN al proprio interno.

Il testo originale esegue l'eliminazione dei simboli di EOLN semplicemente così :

```
if (title[i] == '\n')
  title[i] = ' ';
```

Il problema nasce dal fatto che il file è stato aperto con l'attributo `binary`: la comparazione di `title[i]` con `\n` elimina solo il carattere `LF`, mentre bisogna eliminare anche il `CR`. Il codice va dunque modificato come segue:

```
if ((title[i] == '\n') || (title[i] == '\r'))
  title[i] = ' ';
```

6.1.3 Correzione di un errore

Esattamente nella medesima funzione `parsetitle(char *)` descritta al paragrafo precedente è stato localizzato ed eliminato un errore nel funzionamento di **SWISH**.

Dopo aver sostituito tutti gli EOLN con spazi, bisogna procedere all'eliminazione dei caratteri *bianchi* in testa ed in coda alla stringa.

Questo compito è svolto dai seguenti due cicli `for`:

```
for (i = 0; isspace(title[i]) || title[i] == '\"'; i++)
  ;

for (j = 0; title[i]; j++)
  title[j] = title[i++];
```

In questo punto sono stati eliminati gli spazi iniziali, il valore della stringa è stato semplicemente "traslato verso sinistra": ad esempio, se la stringa fosse stata ``  My Home Page  ''``, il risultato sarebbe ``My Home Pageage  ''``. Si noti come la parte terminale della stringa non sia stata modificata.

L'errore esiste in quanto la prossima istruzione eseguita è una chiamata a `strlen(title)`, che non può ritornare il valore corretto, in quanto la stringa non è stata terminata dopo la traslazione:

```
for (
  j = strlen(title) - 1;      /* errore! */
  (j && isspace(title[j])) || title[j] == '\0' || title[j] == '\"';
  j--
)
  title[j] = '\0';
```

L'errore è risolto terminando la stringa dopo aver eliminato gli spazi iniziali. Il codice completo corretto è il seguente:

```
for (i = 0; isspace(title[i]) || title[i] == '\"'; i++)
  ;

for (j = 0; title[i]; j++)
  title[j] = title[i++];

title[j] = '\0';      /* terminazione della stringa */

for (
  j = strlen(title) - 1;
  (j && isspace(title[j])) || title[j] == '\0' || title[j] == '\"';
  j--
)
  title[j] = '\0';
```

6.1.4 Carattere separatore tra i file

Esiste un'altra importante differenza fra gli ambienti Unix e Dos/Windows: il carattere usato per separare i direttori nel nome di un file: `/` in Unix, \ in Windows.

SWISH è un programma che esegue parecchie elaborazioni su nomi di file e di direttori, quindi in linea di principio bisognerebbe modificarne il codice in ogni punto in cui questa differenza potrebbe causare un diverso funzionamento. Windows (e anche Dos) ha però la particolarità di poter interpretare anche il carattere `/` come separatore fra direttori nel nome di un file[1].

[1]In realtà, \ è necessario nella linea di comando in quanto `/` è usato per separare gli argomenti da passare al programma; all'interno di un programma si può però indicare un file usando come separatore di direttori indifferentemente `/` o \.

6.2 Arricchimento dello *spider*

Come spiegato al capitolo precedente, per la realizzazione della componente di *spider* del sistema per il recupero delle informazioni si dispone del sorgente Java della libreria di classi Acme e dell'applicazione esemplificativa WebCopy.

Lo *spider* viene utilizzato unicamente durante il processo di aggiornamento degli indici contenuti nella BASE DI CONOSCENZA. Esso deve quindi soddisfare le specifiche descritte nella sezione 5.2.3. Dal punto di vista realizzativo, si è proceduto alla modifica del codice di WebCopy, studiandone il comportamento, eliminando le caratteristiche superflue, introducendo le funzionalità richieste: questo nuovo programma è stato chiamato **webcp**.

6.2.1 Linea di comando

La prima caratteristica differente fra **webcp** e WebCopy risiede nei parametri da specificare sulla linea di comando.
WebCopy offre infatti diverse opzioni, molte delle quali sono inutili nella realizzazione della componente di *spider* dell'applicazione per il recupero delle informazioni da sviluppare. Queste sono le opzioni fondamentali da specificare a WebCopy:

- `-f`: se il documento da recuperare corrisponde ad un file già esistente sul disco rigido locale, lo sovrascrive senza cambiargli nome.
  In **webcp** è un'opzione inutile poichè, almeno nel funzionamento ideale del sistema, il direttorio in cui saranno trasferiti i documenti sarà sempre vuoto;

- `-d`: determina il massimo numero di riferimenti ipertestuali da seguire nel reperimento dei documenti. Quest'opzione è inutile, in quanto il parametro può essere impostato accanto al nome del documento da recuperare nel file che descrive la BASE DI CONOSCENZA (vedere il paragrafo A.2).

La linea di comando con cui richiamare **webcp** ha invece una sintassi ben determinata:

```
java webcp <siti> <dir> <ext> [ report | spider ]
```

- `<siti>`: è il nome di un file di testo contenente l'elenco dei nomi (le URL) dei documenti da scaricare, uno per linea.
  Questo elenco viene passato attraverso un file per agevolare la comunicazione con il programma di aggiornamento dell'indice: esso scrive un file con l'elenco dei documenti associati ad ogni categoria, rendendolo

disponibile allo *spider*. L'alternativa a questa soluzione consisterebbe nell'elencare tutti i documenti sulla riga di comando; considerando però che le categorie non hanno limiti al numero di documenti di cui sono composte, questo avrebbe potuto portare ad un superamento dei limiti della memoria disponibile per la linea di comando.

- `<dir>`: è il direttorio in cui **webcp** dovrà trasferire il contenuto dei testi recuperati.
  Al di sotto di esso saranno ricostruite le strutture di direttori originali per ogni documento recuperato, includendo anche il nome del server con cui si è aperta la connessione.

- `<ext>`: è una stringa che contiene l'elenco delle estensioni ammissibili per i documenti da recuperare, separate da spazi.
  Questa opzione non è strettamente necessaria, ma può essere usata per evitare malfunzionamenti, imponendo allo *spider* di prendere in considerazione solo documenti di tipo HTML o comunque testuale.

- `[report | spider]`: è il parametro che determina l'effettivo funzionamento di **webcp** come copiatore del contenuto dei documenti (`spider`) o come elencatore dei loro nomi con attributi di dimensione e data di ultima modifica (`report`).

6.2.2 Gestione report

La modifica concettualmente più importante apportata a WebCopy è dotarlo della possibilità di eseguire "rapporti" (cioè elencare i nomi dei file con gli attributi di dimensione e data di ultima modifica) a partire dai documenti visitati, piuttosto che recuperarne il contenuto.

Da un punto di vista algoritmico, questa possibilità si realizza non aprendo una connessione con la URL associata al documento da visitare, ma limitandosi ad estrarne gli attributi dall'istanza (detta `uc`) della classe `URLConnection`:

```
long mod = uc.getLastModified(); // data di ultima modifica
int len = uc.getContentLength(); // dimensione
```

6.2.3 Nomi dei documenti recuperati

Un'altra modifica da apportare a WebCopy è fargli ricostruire sul disco rigido locale la medesima struttura di direttori in cui il documento da recuperare si trova sul sito remoto, includendo anche il nome del server in cui si trova.

WebCopy memorizza nella variabile `localName` il nome locale del file corrispondente al documento che si sta recuperando: esso consiste nel nome originale del documento (`thisUrlStr`), privato del nome del server e della serie di direttori specificati:

```
thisUrlStr = "http://www.server.it/dir1/dir2/testo.html"
```

verrà copiato localmente come:

```
localName = thisUrlStr.substring(baseUrlStr.length())
```

che produce `localName = ''testo.html''` nel direttorio corrente.

Il funzionamento corretto secondo le specifiche è invece quello prodotto dalla seguente istruzione:

```
String localName =
  outDir +
  thisUrlStr.substring(thisUrlStr.indexOf("://") + 3);
```

Cioè si memorizza in `localName` l'intero nome della URL del documento, privandola solo dell'indicazione del nome del protocollo usato per la connessione; questo archivio viene posizionato nel direttorio specificato da `outDir`, corrispondente al parametro `<dir>` passato sulla linea di comando. Nell'esempio precedente, se `outDir` fosse `c:/database/`, verrebbe creato un file dal nome:

```
c:/database/www.server.it/dir1/dir2/testo.html
```

Dal codice sorgente originale vengono anche eliminati tutti i controlli sull'eventuale preesistenza del file che si sta creando; cioè se `localName` corrisponde ad un file esistente, esso viene riscritto.
Questo comportamento non è affatto restrittivo, ed è anzi pienamente in linea con le specifiche date dal BLOCCO DI AGGIORNAMENTO: quest'ultimo, infatti, cancella il contenuto del direttorio corrispondente ad `outDir` prima di eseguire **webcp**. Quindi eventuali conflitti di nomi tra file già presenti nel direttorio `outDir` e `localName` sarebbero causati unicamente da errori nell'elenco delle pagine HTML associate ad una categoria, nel caso in cui la stessa pagina venisse ripetuta più volte.

6.2.4 Limite al dominio in cui si trovano i documenti

WebCopy consente di recuperare pagine HTML correlate da riferimenti ipertestuali solo nel caso in cui esse si trovino nei direttori "figli" (cioè gerarchicamente inferiori nella struttura ad albero dei direttori) rispetto alla pagina iniziale. Ad esempio, se la pagina

```
http://www.server.it/padre/figlio/testo.html
```

ha un riferimento ipertestuale verso

```
http://www.server.it/padre/riferito.html
```

quest'ultima pagina non viene presa in considerazione dallo *spider*.

Dal punto di vista realizzativo, questo significa verificare che la parte iniziale del nome della URL che è stata estratta dal riferimento ipertestuale (`thisUrlStr`) coincida con la parte iniziale della URL della pagina che è stata visitata (`baseUrlStr`); il metodo `doThisUrl` nella classe `WebCopySpider` decide se visitare la URL `thisUrlStr` o ignorarla:

```
protected boolean doThisUrl(
  String thisUrlStr, int depth, String baseUrlStr
) {
  if ( thisUrlStr.startsWith( baseUrlStr ) &&
    ( parent.maxDepth == -1 || depth <= parent.maxDepth ) )
    return true;
   return false;
 }
```

Dal punto di vista di **webcp**, questo vincolo è stato rilasciato; il metodo `doThisUrl` andrà dunque modificato come segue:

```
protected boolean doThisUrl(
  String thisUrlStr, int depth, String baseUrlStr
) {
  if (
    ( parent.maxDepth == -1 || depth <= parent.maxDepth )
  )
    return true;
   return false;
 }
```

6.2.5 Modifica della libreria Acme

Nel suo funzionamento originale, la libreria Acme realizza le funzionalità di uno *spider* nel seguente modo: dato il nome (URL) di una pagina iniziale, viene trasferita sulla macchina locale (recuperata), e da essa vengono estratti i riferimenti ipertestuali alle pagine correlate, i quali vengono inseriti in una

coda di pagine da visitare; il recupero continua fino all'esaurimento della coda.

Per ogni riferimento ipertestuale estratto, viene inserita nella coda la pagina corrispondente unitamente all'elenco dei suoi direttori gerarchicamente superiori, fino a raggiungere la radice della struttura. Ad esempio, se la pagina HTML

```
http://www.server.it/padre/figlio/testo.html
```

contiene un riferimento a

```
http://www.server.it/padre/figlio/nipote/riferito.html
```

nella coda delle pagine da visitare vengono aggiunte:

```
http://www.server.it/padre/figlio/nipote/riferito.html
http://www.server.it/padre/figlio/nipote/
http://www.server.it/padre/figlio/
http://www.server.it/padre/
http://www.server.it/
```

Questo comportamento produce due effetti:

- vengono messe in coda indistintamente tutte le pagine presenti nel sito che risultino raggiungibili seguendo riferimenti ipertestuali;
- molte pagine possono trovarsi inserite in coda più volte; il loro recupero è eseguito una sola volta facendo un uso opportuno di una `Hashtable`.

L'inserimento in coda è determinato dal metodo `add` della classe `Spider`:

```
private void add( String urlStr, URL contextUrl, SpiderItem item )
    {
    try
        {
        URL url = Acme.Utils.plainUrl( contextUrl, urlStr );
        urlStr = url.toExternalForm();
        addOne( urlStr, contextUrl, item );

          /*
            \'e in questo ciclo while che vengono
            inseriti i direttori ``padri''
          */
        String rootUrlStr =
```

```
            ( new URL( new URL( urlStr ), "/" ) ).toExternalForm();
        while ( urlStr.length() > rootUrlStr.length() )
            {
            int lastSlash = urlStr.lastIndexOf( '/', urlStr.length() - 2 );
            urlStr = urlStr.substring( 0, lastSlash + 1 );
            addOne( urlStr, contextUrl, item );
            }
        }
    catch ( MalformedURLException e )
        {
        String msg = e.getMessage();
        if ( checkMalformedURL( msg ) )
            brokenLink( myUrlToString( contextUrl ), urlStr, msg );
        }
    }
```

Dal punto di vista del funzionamento di **webcp**, questo comportamento è inutile se non dannoso. Nella maggior parte dei casi, infatti, le categorie sono costituite da singole pagine HTML, ed è sbagliato trasferire localmente (e quindi successivamente anche indicizzare) tutte le informazioni dei siti in cui risiedono.
La modifica va chiaramente apportata al metodo `add` della classe `Spider` precedentemente descritto. La soluzione da adottare appare molto chiara:

- Java è un linguaggio orientato ad oggetti che consente l'ereditarietà;
- Acme è una libreria di classi scritta in Java;
- **webcp** definisce una classe dal nome `WebCopySpider` che estende (eredita da) la classe `Spider` definita nella libreria;
- quindi basta ridefinire il metodo `add` all'interno della classe `WebCopySpider`.

Ebbene, questo procedimento è proibito in quanto il metodo `add` è stato dichiarato `private` nella classe `Spider`:

```
private void add( String urlStr, URL contextUrl, SpiderItem item )
```

Le soluzioni alternative sono le seguenti:

- costruire un ulteriore modulo di programma, identico a `Spider.java`, tranne che per il fatto di contenere la versione opportunamente modificata del metodo `add`, e fare estendere questo nuovo modulo a `WebCopySpider`.

Questa soluzione sarebbe una negazione della filosofia della programmazione ad oggetti; l'ereditarietà è un concetto fondamentale della programmazione moderna (vedere il capitolo 4), introdotto proprio come soluzione al problema dell'estensione del codice, per evitare dannose riscritture e maldestre modifiche: è *sbagliato* avere due moduli simili all'interno dello stesso programma;

- modificare direttamente il codice di `add` nella libreria Acme, variandone permanentemente il funzionamento. Questa soluzione sarebbe invece una negazione del concetto di libreria: esse esistono per fornire strumenti di supporto al programma, ma è sbagliato modificarne il contenuto; piuttosto se ne devono creare di nuove;

- modificare la dichiarazione del metodo `add` all'interno della libreria, così da consentirne la ridefinizione in una classe (`WebCopySpider`, in particolare) che erediti da `Spider`. La soluzione adottata è effettivamente questa. Si noti che anche in questo caso la modifica coinvolge la libreria (che, per quanto detto al punto precedente, non andrebbe modificata); a differenza di prima, però, la modifica è sostenuta dai seguenti tre argomenti:

  - è la soluzione che comporta la minor quantità di codice riscritto, in quanto non c'è una ricopiatura dell'intera classe `Spider` ma solo inevitabilmente del corrispondente metodo `add`;
  - è la soluzione che comporta il minor intervento sul codice della libreria, in quanto si modifica semplicemente la dichiarazione di un metodo, e non già il suo comportamento;
  - la decisione di intervenire sul codice della libreria in questa maniera è stata presa tenendo in considerazione anche quanto autorevolmente scritto da B. Meyer[20]: una classe dovrebbe essere sempre aperta per future estensioni, il che implica che non è compito suo decidere quali metodi possono essere ridefiniti dai suoi discendenti. Tenendo presente ciò, è addirittura possibile affermare che l'*errore* è presente nella libreria, e questa modifica lo elimina.

Dal punto di vista degli interventi sul codice, vanno dunque fatte le seguenti modifiche:

- nuova dichiarazione del metodo `add` della classe `Spider` nel testo sorgente di nome `Spider.java` nella libreria Acme:

```
public class Spider implements Acme.HtmlObserver, Enumeration {
```

```
  ...
  protected void add( String urlStr, URL contextUrl, Acme.SpiderItem item )
  {
    ...
  }
}
```

- nel codice di **webcp**, nella classe `WebCopySpider` che eredita da `Spider`, ora è possibile ridefinire il metodo `add`. Rispetto alla versione precedente è da eliminare il ciclo `while` che eseguiva le `addOne` su tutti i direttori di livello gerarchicamente superiore a `urlStr`:

```
protected void add( String urlStr, URL contextUrl, Acme.SpiderItem item )
{
  try
    {
    // Convert to no-ref, canonical form.
    URL url = Acme.Utils.plainUrl( contextUrl, urlStr );
    urlStr = url.toExternalForm();

    addOne( urlStr, contextUrl, item );
      /* qui c'era il while */
    }
  catch ( MalformedURLException e )
    {
    String msg = e.getMessage();
    if ( checkMalformedURL( msg ) )
        brokenLink( myUrlToString( contextUrl ), urlStr, msg );
    }
}
```

6.2.6 Gestione delle estensioni

L'ultima modifica presa in esame in questa sezione riguarda la possibilità di specificare un elenco di estensioni ammissibili per i file da visitare.
Ad esempio, per limitare il trasferimento alle sole informazioni di tipo testuale da un determinato sito, si può specificare nel parametro `<ext>`: ``.html

`.htm .txt .pdf''`. Verranno recuperati solo i documenti con le estensioni indicate.

Per realizzare una tale funzione, si è proceduto allo sviluppo del seguente metodo `isGood` nella classe `webcp`, in cui il parametro formale `s` è il nome completo (URL) del testo da recuperare di cui si vuole testare l'estensione, mentre la variabile `myExtensions` è il valore del parametro `<ext>`:

```
protected boolean isGood(String s) {
    // is a directory?
  if (s.endsWith("/"))
    return true;

    // check myExtensions
  else {
    boolean found = false;
    StringTokenizer stream = new StringTokenizer(myExtensions);

    while (!found && stream.hasMoreTokens())
        found = s.endsWith(stream.nextToken());

    return found;
  }
}
```

La condizione

```
if (isGood(thisUrl))
```

va verificata nel momento in cui si dispone del nome completo della URL, quindi subito dopo aver aperto la connessione.

6.3 Modifica dell'interfaccia in JavaScript

Per la realizzazione del BLOCCO DI INGRESSO è stato utilizzato il codice sviluppato da Marcelino Martins e reso pubblico all'indirizzo

`http://www.geocities.com/Paris/LeftBank/2178/`

Tale codice consiste in un programma scritto in JavaScript (vedere il paragrafo 4.3.1) che consente la visualizzazione di un albero rappresentando ogni nodo con una cartella (il tentativo grafico è di riprodurre il medesimo

aspetto dell'albero dei direttori nel FileManager di Windows); cliccando su una cartella questa si apre, rendendo visibili le proprie sottocartelle.

Questo programma viene dunque utilizzato, nell'applicazione sviluppata, per fornire all'utente una visione chiara ed immediata della struttura della BASE DI CONOSCENZA, associando una categoria ad ogni cartella. L'utente può esaminare e selezionare i vari argomenti direttamente cliccando sulle categorie.

Per poter soddisfare le specifiche imposte sul BLOCCO DI INGRESSO nel capitolo precedente, il codice JavaScript è stato sostanzialmente completamente riscritto; in questo contesto non verranno cioè esaminate in dettaglio tutte le modifiche ad esso apportate, ma solo descritte le sue parti più significative.

Strutture ad albero

A differenza delle strutture ad albero che saranno dettagliatamente descritte nel capitolo successivo, questo programma JavaScript memorizza gli alberi nel seguente modo: ogni nodo[2] è un array, i cui elementi hanno il seguente significato:

- `nodo[0], nodo[1]` indicano se questa cartella è aperta o chiusa;
- `nodo[2]` è il numero di figli che sono documenti; nella rappresentazione della BASE DI CONOSCENZA questo valore è sempre 0;
- `nodo[3]` è l'etichetta della cartella, ovvero il nome esterno della categoria, che viene visualizzato all'utente;
- `nodo[4]` è il numero totale delle sottocartelle;
- `nodo[5]` nome interno della categoria: sostanzialmente è la *chiave primaria* dell'albero, il valore mediante il quale le categorie vengono distinte;
- `nodo[6]` indica se questa cartella è stata selezionata dall'utente, per essere inclusa nella ricerca;
- gli elementi di indice maggiore di 6 sono i nodi corrispondenti alle sottocartelle.

[2]Nella presente discussione saranno considerati sinonimi i termini "nodo", "categorie" e "cartella".

Al termine del caricamento del codice JavaScript nel *browser*, l'operazione da compiere prima della visualizzazione dell'albero è la sua inizializzazione con i valori corrispondenti alla BASE DI CONOSCENZA. La funzione `generatetree()` serve proprio a questo scopo.

È importante far notare come il codice effettivo di tale funzione sia automaticamente generato da un programma apposito (`zDump`): funzionalmente, esso legge la descrizione della BASE DI CONOSCENZA dal file `z.cat` e produce in uscita il codice della funzione `generatetree`.

Selezione delle categorie

L'utente può esplorare la BASE DI CONOSCENZA aprendo e chiudendo le categorie con un click del mouse. Come descritto nel paragrafo 5.2.4, egli deve anche poterne selezionare alcune, che verranno usate come parte della chiave di ricerca.

La selezione di una categoria avviene cliccando su di essa quando è aperta. La deselezione di una categoria avviene cliccando su di essa quando è selezionata (questo comporta in realtà anche la sua chiusura). Visivamente, le categorie selezionate sono evidenziate in grossetto e con un colore particolare.

Il codice che gestisce la selezione e deselezione delle categorie si trova nella funzione ricorsiva:

```
function clickOnFolderRec(foldersNode, folderName)
```

dove:

- `foldersNode` è il nodo corrente da esaminare;
- `folderName` è il nome interno della categoria su cui l'utente ha cliccato;

Se il nome interno della categoria associata al nodo corrente è diverso dal nome che è stato cliccato, `clickOnFolderRec` si richiama ricorsivamente su ogni nodo figlio del corrente:

```
for (i = FIELDNO ; i < foldersNode.length  - foldersNode[2]; i++)
  clickOnFolderRec(foldersNode[i], folderName, 1)
```

Se i due nomi coincidono:

```
if ((foldersNode[0] == 1) || (foldersNode.length == FIELDNO)) {
  if (foldersNode[6] == 0)
    foldersNode[6] = 1;
  else {
    foldersNode[6] = 0;
```

```
        closeFolders(foldersNode)
      }
    } else {
      foldersNode[0] = 1
      foldersNode[1] = 1
    }
```

cioè:

- se la categoria è aperta o non ha figli, la si seleziona o deseleziona e chiude;
- altrimenti la si apre.

Estrazione dei riferimenti

L'insieme delle categorie selezionate deve essere inviato al BLOCCO DI RICERCA come parte della chiave di ricerca espressa dall'utente.

Per comprendere come ciò avvenga è necessario descrivere la form HTML che l'utente compila; essa è qualcosa del tipo:

```
<form
  name="viceform" method=post
  action=http://www.eurotalete.com/servlet/z
  onSubmit="toSearch()"
>
  <input type=text   name="kwd"    value="">
  <input type=hidden name="cat"    value="">
  <input type=hidden name="switch" value="search">
  ...
</form>
```

cioè:

- i campi della form verranno inviati al programma di nome

  `http://www.eurotalete.com/servlet/z`

  col metodo `post`;
- la form consta di tre campi:

  - `kwd` è il campo di testo contenente le parole chiave digitate dall'utente;
  - `switch` è un campo che consente al BLOCCO DI RICERCA di capire che lo si sta utilizzando come estrattore di riferimenti e non per sottoporre una domanda all'esperto in linea;
  - `cat` è il campo che contiene l'elenco dei nomi interni delle categorie selezionate.

- quando l'utente richiede la sottomissione della form deve essere richiamata la funzione `toSearch()` del codice JavaScript.

È proprio quest'ultima funzione che ha il compito di scrivere nel campo `cat` della form l'elenco dei nomi interni delle categorie selezionate. Il codice di `toSearch` è il seguente:

```
function toSearch() {
  cattmp = "";
  dumpSelected(foldersTree, 0, 1);
  if (cattmp == "")
    selectAll(foldersTree);
  top.frames[1].document.viceform.cat.value = cattmp;
}
```

che fondamentalmente si limita a scrivere in `cat` il valore della variabile `cattmp`; quest'ultima è inizializzata dalla funzione `dumpSelected`, che ricorsivamente visita l'albero delle categorie aggiungendo alla variabile `cattmp` i nomi interni di quelle selezionate:

```
function dumpSelected(myroot, found, dumpSubnode) {
var n = 0

  if (myroot != 0) {
    if  ((dumpSubnode && found) || myroot[6]) {
      if (dumpSubnode == 1)
        found = 1;

      cattmp += " " + myroot[5];
    }

    for (n = FIELDNO; n < myroot.length; n++)
      dumpSelected(myroot[n], found, dumpSubnode);
  }
}
```

Limiti di questo approccio

Concludendo, è possibile affermare che l'uso di questo codice JavaScript reca notevoli vantaggi: è semplice da capire e può essere adattato a qualunque struttura di albero, è direttamente interpretato dai *browser* e non necessita quindi di alcun programma aggiuntivo per essere eseguito.

D'altra parte sono numerosi anche i suoi limiti, in particolare tenendo conto del fatto che la struttura della BASE DI CONOSCENZA è destinata ad aumentare, in quanto nuovi documenti periodicamente verranno aggiunti nella BASE DI DATI.

Il limite più appariscente è la lentezza con cui l'albero reagisce ai comandi impartiti dall'utente, ad esempio per l'apertura o chiusura delle cartelle. Questa lentezza non potrà che aumentare con una struttura ad albero più ramificata.

Un limite che non riguarda la situazione odierna è invece dovuto al fatto che la dimensione (in byte) della funzione `generatetree` è strettamente correlata e piuttosto simile a quella del file `z.cat`. Poichè la BASE DI CONOSCENZA, come già detto, è destinata ad arricchirsi con nuovi documenti, la dimensione di `z.cat` e conseguentemente del codice JavaScript è destinata a crescere. Dal momento che tale codice deve essere scaricato dalla rete da parte di un utente, non può crescere smisuratamente, pena un insopportabile ritardo tra l'inizio della connessione e la visualizzazione dell'albero. L'approccio basato sull'uso di JavaScript non consente alcun tipo di soluzione. L'approccio basato sull'uso di Applet Java sì: si veda il capitolo 8 per una discussione su questo tema.

In definitiva, questo codice non è ulteriormente estendibile, rappresenta il massimo che si possa ottenere usando JavaScript.

Capitolo 7

Software prodotto

Il codice sviluppato per l'applicazione di recupero delle informazioni non locali da realizzare può essere suddiviso nei seguenti programmi:

- `z`: è la Servlet che esegue l'estrazione dei riferimenti dalla BASE DI DATI; utilizza **swish**;

- `zAnsw`: è la Servlet che realizza il BLOCCO DI RISPOSTA dell'esperto (vedere fig. 5.2);

- `Update`: è il modulo che realizza il BLOCCO DI AGGIORNAMENTO della BASE DI DATI; utilizza **swish**, integra **webcp** e la libreria Acme, ed i programmi di utilità per cancellare, copiare, comparare file;

- `zDump`: è un programma di supporto alla fase di configurazione del sistema: ad esempio controlla la correttezza sintattica del file `z.cat` ed è in grado di generare i file HTML che visualizzano la struttura ad albero; esso non verrà ulteriormente discusso, in quanto non fa espressamente parte del sistema informativo, è solo uno strumento di supporto per la fase di sviluppo;

- `zzz`: è un'Applet che realizza il BLOCCO DI INGRESSO ed il BLOCCO DI USCITA in modo alternativo rispetto al codice JavaScript ed HTML che è stato utilizzato in questa tesi;

Per il loro funzionamento, questi programmi necessitano di una serie di moduli comuni, che sono stati inseriti in due pacchetti (`package`) Java:

- `vice`: contiene una serie di classi di utilità generale, riutilizzabili e svincolate dalla particolare applicazione oggetto di questa tesi;

- `zuf`: contiene le classi dipendenti da questa particolare applicazione di recupero delle informazioni.

Ora verranno descritti dettagliatamente tutti i componenti sopracitati, utilizzando un approccio "bottom-up": saranno descritti per primi quelli il cui funzionamento è indipendente dagli altri.

7.1 Pacchetto `vice`

Questo pacchetto è una raccolta di classi che forniscono strumenti di uso assolutamente generale e quindi riutilizzabili anche in altre applicazioni. La loro realizzazione si è resa necessaria durante le sviluppo delle altre parti del sistema e sono state inserite qui.

7.1.1 Gestione dei file di template

Un file di "template" è un documento il cui contenuto viene parzialmente deciso dal programma in fase di esecuzione, generalmente mediante sostituzione testuale di particolari linee di testo. Possiamo chiamare per comodità "etichette" tali particolari linee del file di template e "valore effettivo" il valore che viene sostituito al posto dell'etichetta.

Questo tipo di documenti sono usati, nell'applicazione sviluppata, per rendere più flessibili e facilmente configurabili i testi HTML prodotti dalle Servlet: quando, ad esempio, il BLOCCO DI RICERCA (vedere il paragrafo 7.3) deve produrre il documento con l'elenco dei riferimenti corrispondenti alla chiave di ricerca, il template è qualcosa del tipo:

```
<html>
  <head>
    <title>Risultati della ricerca</title>
  </head>
  <body>
    Chiave di ricerca:
    [KWD]
    <br>Risultati:<br>
    [RESULT]
  </body>
</html>
```

La struttura del documento prodotto sarà cioè quella indicata, mentre i valori effettivi della chiave di ricerca e dei riferimenti trovati sono noti al programma

in fase di esecuzione e verranno sostituiti al posto delle "etichette" `[KWD]` e `[RESULT]`.

Tale meccanismo risulta anche fondamentale per assicurare che i testi prodotti dalle Servlet siano *visivamente omogenei* con le altre pagine HTML che compongono il servizio offerto sul sito.

Dal punto di vista del codice Java, un file di template viene gestito con l'ausilio dei seguenti strumenti:

- la classe `Templater`.
  Gli oggetti istanze di tale classe memorizzano:

  - un riferimento al file di template sul disco rigido;
  - un riferimento ad un elenco di oggetti istanze di classi che implementano l'interfaccia `Generator`, sostanzialmenti corrispondenti alle diverse "etichette" presenti in questo file di template.

  Quando viene richiamato il metodo di generazione del template (metodo `go`), ogni linea del file di template viene controllata rispetto a tutti gli oggetti generatori: se viene trovata una corrispondenza è generato il corrispondente "valore effettivo", altrimenti viene semplicemente stampata la linea di testo.

- l'interfaccia `Generator`.
  Gli oggetti, istanze di classi che implementano tale interfaccia, realizzano in pratica l'associazione fra "etichetta" e "valore effettivo".
  Tale associazione è ottenuta con i seguenti due metodi:

  - `matches`: indica se la linea di testo del file di template passata come argomento corrisponde all'etichetta incapsulata in questo oggetto generatore;
  - `explode`: ritorna il "valore effettivo", effettuando tutte le computazioni necessarie alla determinazione di tale valore.

Nel caso in cui il "valore effettivo" fosse una semplice stringa di testo costante, può efficacemente essere usata la classe `SimpleGen`. Ad esempio, per realizzare l'associazione fra l'etichetta `[TITOLO]` ed il valore `"Risultati della ricerca"` il codice da usare è il seguente:

```
new SimpleGen("[TITOLO]", "Risultati della ricerca")
```

7.1.2 Problemi di condivisione di risorse

Per la risoluzione dei problemi causati dall'esecuzione concorrente di più processi[1] è stata creata la classe `Sync`; nella presente applicazione per il recupero delle informazioni sono due i casi in cui possono verificarsi conflitti per l'accesso contemporaneo a risorse.

Sostituzione di un indice obsoleto con uno testè generato da parte del BLOCCO DI AGGIORNAMENTO

I problemi sorgono quando il BLOCCO DI AGGIORNAMENTO sta compiendo l'aggiornamento di un indice nella BASE DI CONOSCENZA mentre un utente sta concorrentemente usando il sistema informativo, ed in particolare ha attivato una ricerca che comprenda anche l'indice in questione: senza un'opportuna gestione, infatti, può capitare che l'estrazione dei riferimenti avvenga da un file di indice parziale ed in generale corrotto, in quanto il processo di aggiornamento non ne ha ancora terminato la scrittura.

La soluzione adottata è scomponibile in due fasi:

1. scrittura (da parte evidentemente del BLOCCO DI AGGIORNAMENTO) del nuovo indice con un nome diverso dall'originale;

2. copia *sicura* del nuovo indice su quello obsoleto. Questa fase corrisponde esattamente al metodo `copy` della classe `Sync`. Esso compie le seguenti operazioni:

   (a) cancellazione dell'indice obsoleto. È questa operazione che consente la *sincronizzazione*: se infatti un altro processo (quello di estrazione dei riferimenti, evidentemente) sta mantenendo aperto il file, il sistema operativo Windows ne impedisce la cancellazione. Solo dopo la chiusura del file può avvenirne la cancellazione.

   (b) ridenominazione del nuovo indice col nome originale.

   Si noti come queste due operazioni siano entrambe *atomiche* (cioè non interrompibili da un altro processo): è questa atomicità che rende possibile la sincronizzazione del funzionamento di più processi contemporaneamente.

[1] È necessario fare una importante distinzione: in questo paragrafo si affrontano e risolvono problemi causati dall'esecuzione contemporanea di più *processi* intesi come programmi, a livello del sistema operativo.
Questi problemi sono ben diversi rispetto a quelli causati dall'esecuzione contemporanea di più *thread* all'interno del medesimo programma.

Da un punto di vista realizzativo, la copia *sicura* del file `sorgente.txt` sul file `destinazione.txt` si effettua con la seguente chiamata:

```
Sync.copy("sorgente.txt", "destinazione.txt");
```

Memorizzazione della domanda posta da un utente all'esperto

I problemi sorgono quando due utenti stanno concorrentemente inviando due domande col medesimo titolo: essendo il nome del file che memorizza la domanda costituito dalla data e dal titolo (per la spiegazioni di tali dettagli vedere il paragrafo 7.2.4), entrambi i processi aprirebbero il medesimo file per scrivere due testi diversi.

La soluzione adottata è piuttosto drastica: la scrittura nel direttorio QUEST (in cui sono memorizzate appunto le domande poste dagli utenti) è consentita ad un solo processo per volta.

La soluzione adottata per tale sincronizzazione è la seguente:

- nel direttorio QUEST esiste un file, che è possibile chiamare "semaforo"; tale file deve essere preesistente alla prima esecuzione del programma: evidentemente va creato in fase di configurazione e installazione del sistema;

- il processo che è in grado di cancellare il "semaforo" ha il diritto di scrivere sul direttorio;

- è fondamentale che (qualunque cosa succeda), al termine delle operazioni sul direttorio tale processo ripristini il "semaforo": se così non fosse, nessun altro potrebbe più arrogarsi il diritto di accesso a quella risorsa, cioè la scrittura sul direttorio QUEST.

Dal punto di vista realizzativo, le operazioni da compiere sono fornite dalla classe `Sync` sono semplicemente le seguenti:

1. la chiamata *bloccante*, che mette cioè il processo in attesa del diritto di accedere alla risorsa, si effettua creando una nuova istanza della classe `Sync`, a cui va indicato il nome del "semaforo":

   ```
   Sync unSemaforo = new Sync("c:\\quest", "semaforo.txt");
   ...
   ```

quando il flusso di esecuzione giunge al codice contraddistinto da ..., significa che l'accesso alla risorsa è consentito: il processo corrente è l'*unico* che può scrivere sul direttorio;

2. al termine delle operazioni il "semaforo" va ristabilito chiamando il metodo `reset` oppure `release`:

```
unSemaforo.reset();
```

7.1.3 Gestione delle strutture ad albero

Il tipo di dato fondamentale nel funzionamento dell'applicazione per il recupero delle informazioni oggetto di questa tesi è la struttura ad albero.

Una struttura ad albero viene normalmente definita ricorsivamente come segue (si faccia riferimento alla figura 7.1):

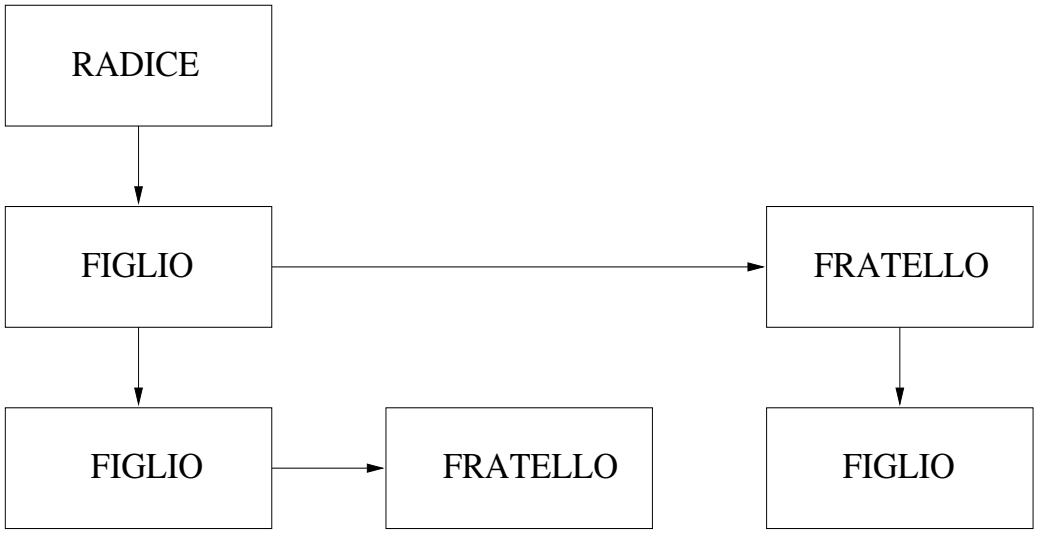

Figura 7.1: Struttura ad albero

- l'albero ha una radice, che è un nodo;
- ogni nodo ha un riferimento al sottoalbero dei nodi figli, ed uno all'albero dei nodi fratelli;
- ogni nodo ha un riferimento ad un oggetto che mantiene memorizzate le informazioni associate a quel nodo; possiamo chiamare tale oggetto "valore" del nodo.

Questa struttura è definita nella classe `Node`, con un codice simile al seguente:

```
class Node {
  Node brother, child;
  Linkable obj;        // valore del nodo
  ...
}
```

Tutti i tipi di alberi necessari nell'applicazione devono condividere tale codice. Essi possono differenziarsi l'un l'altro per avere tipi di nodi e/o oggetti "valore" diversi. Ad esempio, nell'applicazione sviluppata sono costruiti due diversi alberi (le classi citate verranno dettagliate nel prosieguo del capitolo):

- quello preposto alla memorizzazione della struttura della BASE DI CONOSCENZA; esso è basato su un tipo di nodo `RdNode` con un "valore" `Category`;
- quello preposto alla memorizzazione ed alla visualizzazione dei riferimenti estratti dalla BASE DI CONOSCENZA; esso è basato su un tipo di nodo `ONode` con un "valore" `OCategory`.

È quindi necessario prevedere un meccanismo che memorizzi e gestisca l'associazione fra i tipi effettivamente usati all'interno di un albero per i nodi e i valori. Tale meccanismo è realizzato dall'interfaccia `TreeProducer`, che ha un codice simile al seguente:

```
interface TreeProducer {
  Node newNode();
  Linkable newObj(Node n);
}
```

Le classi che implementano tale interfaccia contengono un codice del tipo (l'esempio è preso dalla classe `OTree`, che realizza l'albero dei riferimenti estratti):

```
class OTree implements TreeProducer {
  public Node newNode() { return new ONode(this); }
  public Linkable newObj(Node n) { return new OCategory(n); }
  ...
}
```

In pratica, i metodi che effettivamente creano nuovi nodi e nuovi "valori" (ad esempio il metodo `readFrom` della classe `RdNode` o il metodo `mkChild` della classe `MkNode`) non istanziano direttamente i nuovi oggetti, ma demandano

tale operazione all'oggetto istanza della classe che implementa `TreeProducer` corrispondente a quel particolare albero, che si comporta quindi come un "erogatore" di componenti dell'albero: quando si necessita di un nuovo nodo, si ordina all'"erogatore" (`producer`) di produrne uno con la chiamata `producer.newNode()`; quando si necessita di un nuovo "valore", si ordina all'"erogatore" di produrne uno con la chiamata `producer.newObj(this)`.

La struttura dell'albero viene inoltre completata mantenendo i seguenti altri riferimenti:

- ogni nodo ha un riferimento al `TreeProducer` che deve utilizzare per produrre nuove componenti dell'albero;

- ogni oggetto "valore" all'interno di un nodo ha un riferimento al nodo medesimo.

Discusso il funzionamento complessivo, vengono ora brevemente descritti i metodi messi a disposizione dalle varie classi che realizzano gli alberi nel programma sviluppato.

Classe `Node`

Questa è la classe padre di tutte quelle che realizzano tipi diversi di nodi dell'albero. Essa contiene i metodi generali, applicabili ad ogni albero:

- `find(Obkect key)`.
  Ricerca, all'interno dell'albero che ha come radice questo nodo, un elemento che corrisponda al parametro `key`. La corrispondenza viene testata mediante il metodo `matches` dell'interfaccia `Linkable`, che deve essere implementata da tutti gli oggetti "valore".

- `getChild(Object key)`.
  Ricerca l'elemento corrispondente al parametro `key` nell'insieme dei figli di questo nodo.

- `doAction(Action act)`.
  Consente di eseguire *azioni* estese a tutto il sottoalbero che ha come radice questo nodo. Una *azione* è un oggetto istanza di una classe che implementa l'interfaccia `Action`: i suoi metodi vengono richiamati su ogni oggetto "valore" visitato nel sottoalbero.

Classe `RdNode`

La classe `RdNode` (che eredita da `Node`) rappresenta il tipo di nodo di un albero che sia caricabile da un file su disco (è infatti questo il tipo di nodo dell'albero corrispondente alla struttura della BASE DI CONOSCENZA, costruito in memoria basandosi sul contenuto del file `z.cat`).

Le differenze rispetto alla classe `Node` sono fondamentalmente tre:

- l'"erogatore" di componenti dell'albero deve essere di tipo `TreeRdProducer`, deve cioè essere in grado di fornire anche indicazione sulla sintassi del file da cui caricare l'albero. In sostanza, gli oggetti `RdNode` interrogano il `producer` anche per sapere quali sono le stringhe che delimitano il nodo nel file. Ad esempio, il file potrebbe essere qualcosa del tipo:

```
catbegin nodo1
  catbegin nodo figlio catend
  catbegin nodoFratelloDelFiglio catend
catend
```

  L'effettiva sintassi (`catbegin`, `catend`) è nota al `TreeRdProducer` e non alla classe `RdNode`.

- Gli oggetti "valore" non possono più implementare soltanto l'interfaccia `Linkable`, ma devono fornire un ulteriore metodo (`readFrom`, dell'interfaccia `Readable`), che ne consenta l'inizializzazione con i valori letti dal file. Si veda la classe `Category` (paragrafo 7.2.1) per maggiori dettagli.

- Viene definito un metodo `readFrom` che costruisce l'albero in memoria partendo dalla descrizione che ne è data nel file su disco.

Classe `MkNode`

La classe `MkNode` (che eredita da `Node`) rappresenta il tipo di nodo di un albero i cui rami siano costruibili direttamente a partire da una stringa. Ad esempio, un codice del tipo:

```
unMkNode.mkBranch("a | b | c", "|")
```

costruisce (se non esistono già) il nodo `a`, poi il nodo `b` come suo figlio ed infine il nodo `c` come "nipote" di `a`.
Un'ulteriore chiamata

```
unMkNode.mkBranch("a | d", "|")
```

provocherebbe la creazione di un nodo `d` fratello di `b`.

Rispetto al tipo `Node`, la classe `MkNode` aggiunge i seguenti due metodi:

- `mkChild(Object key)`.
  Se nessun nodo, nell'insieme dei figli del nodo corrente, corrisponde al parametro `key`, allora crea un nuovo figlio, inizializzando il nuovo oggetto "valore" con il parametro `key`.

- `mkBranch(String ramo, String sep)`.
  Costruisce il ramo definito dalla stringa `ramo`, come negli esempi sopra descritti. Gli oggetti "valore" creati da tale operazione vengono inizializzati con i "token" della stringa `ramo` (i "token" sono le sottostringhe della stringa `ramo` delimitate dai caratteri presenti nella stringa `sep`).

Questo tipo di albero viene utilizzato per memorizzare la struttura dei riferimenti estratti dalla BASE DI CONOSCENZA. In particolare, per ogni riferimento estratto viene prodotta una stringa, la quale contiene l'elenco delle etichette associate alle categorie da attraversare per raggiungere quel riferimento all'interno della BASE DI CONOSCENZA, separate da un carattere particolare.
La linea di codice che realizza tale operazione è nella classe `Find`:

```
otree.mkBranch(catalog.labelsOf(dirStream.nextToken(), "|"), "|");
```

dove:

- `otree` è l'albero dei riferimenti;
- `catalog` è la struttura della BASE DI CONOSCENZA;
- `catalog.labelsOf(String c, String s)` ritorna l'elenco delle etichette corrispondenti alla categoria di nome `c`, separate dalla stringa `s`;
- `dirStream.nextToken()` individua il nome interno che identifica univocamente la categoria in cui il riferimento è stato trovato.

7.2 Pacchetto zuf

7.2.1 Classe `Category`

La classe `Category` è il tipo che consente la manipolazione delle informazioni contenute in una categoria della BASE DI CONOSCENZA. Gli oggetti

istanze di questa classe sono oggetti "valore" dell'albero corrispondente alla BASE DI CONOSCENZA.

Una categoria è formata dai seguenti campi, esplicitamente presenti (o comunque calcolabili da) nel file `z.cat`:

- `label`: è l'etichetta;
- `name`: è il nome interno;
- `level`: è il livello dell'albero in cui la categoria si trova (il numero di nodi che la separano dal nodo radice);
- `sites`: è un vettore contenente l'elenco dei documenti associati a questa categoria. Per ogni documento vengono memorizzate le seguenti informazioni (definite nella classe `Site`):
  - `linkDepth`: il numero di riferimenti ipertestuali che è necessario seguire nel recupero dei documenti da indicizzare; questa informazione è utile esclusivamente al programma di *spider* nel BLOCCO DI AGGIORNAMENTO;
  - `label`: è il nome significativo con cui il sistema chiama questo documento; in generale è diverso dal titolo (campo `<title>...</title>`) del documento HTML;
  - `url`: è la URL corrispondente a questo documento.
- `dir`: è il direttorio in cui è memorizzato l'indice corrispondente alla categoria corrente; viene calcolato dopo aver completato il caricamento del file `z.cat`, componendo i `name` delle categorie che compongono il percorso dalla categoria radice alla corrente.

La classe `Category` implementa l'interfaccia `Readable`: il metodo

```
readFrom(AssertTokenizer inStream, String open, String close)
```

ha il compito di inizializzare i campi della `Category` con le informazioni lette dal file associato con `inStream`; le stringhe `open` e `close` sono le parole che appunto determinano con quale sintassi vengono indicati l'inizio e la fine di una categoria.

7.2.2 Gestione del file di configurazione

La classe `Config` consente di definire in un file residente sul disco rigido il valore di alcuni parametri del sistema informativo. Tale file ha nome `z.cfg` e si trova nel direttorio `/zuff/config`.

Le altre classi che compongono il programma possono esaminare il valore dei parametri invocando il metodo

```
Config.get(key)
```

dove `key` rappresenta il nome del parametro.

Dal punto di vista del codice Java, la classe `Config` memorizza le coppie di stringhe (nome, valore) in una `Hashtable`, così da consentirne un rapido reperimento. Il tipo di dato `Hashtable` viene fornito dal linguaggio appositamente per memorizzare associazioni (chiave, valore) fra oggetti.

Per i dettagli sulla sintassi e per l'elenco dei parametri configurabili si consulti l'appendice A.1, in cui si trova la configurazione del sistema funzionante sul sito `www.eurotalete.com`.

7.2.3 Estrazione dei riferimenti

La classe `Find` si occupa dell'estrazione dei riferimenti. Il suo funzionamento ricalca esattamente quanto descritto per il BLOCCO DI RICERCA al paragrafo 5.2.5; il costruttore della classe `Find` è infatti definito nel modo seguente:

```
public Find(Catalog catalog, String dirList, String kwdList) {
  ...
}
```

in cui:

- `catalog` è la struttura della BASE DI CONOSCENZA;
- `dirList` è l'elenco dei direttori corrispondenti alle categorie selezionate dall'utente;
- `kwdList` è l'elenco di parole in base alle quali estrarre i riferimenti ai documenti.

Le operazioni compiute dalla classe `Find` sono le seguenti:

- elaborazione della stringa `kwdList`, eliminando da essa le *stop words* ed eseguendo un algoritmo di *stemming*[37][33] sulle restanti;
- esecuzione del programma (**swish**) che effettivamente estrae i riferimenti dagli indici che compongono la BASE DI CONOSCENZA;
- interpretazione dei risultati prodotti da **swish** e conseguente generazione del testo HTML contenente l'elenco dei riferimenti estratti.

Quest'ultima operazione è senza dubbio la più complessa, e va esaminata più in dettaglio. Per prima cosa è necessario analizzare il testo prodotto da **swish**; esso consiste in:

- due linee in cui è stampata la versione di **swish**; poi, per ogni indice elencato sulla linea di comando:
- un certo numero di linee recanti varie informazioni, sostanzialmente inutili ai fini dell'applicazione, che iniziano col carattere `'#'`;
- l'elenco dei riferimenti estratti, uno per riga; per ogni riferimento viene stampato il valore numerico di *ranking*, il nome del file, il titolo del documento (se è un HTML), la dimensione del documento;
- l'elenco dei riferimenti termina con una linea che inizia col carattere `'.'`.

Il metodo `run` della classe `Find` ha il compito di interpretare tali risultati, costruendo con essi un "albero di riferimenti". Gli oggetti "valore" di tale albero (classe `OCategory`) memorizzano fondamentalmente i seguenti campi:

- il nome "esterno" della categoria;
- un vettore in cui sono contenuti i riferimenti estratti da questa categoria.

Al termine dell'operazione di costruzione dell'albero in memoria si procede alla stampa delle informazioni contenute nello stesso, formattate usando i comandi HTML; in particolare, la struttura ad albero viene mantenuta innestando fra loro più comandi di liste:

```
<du>
  <dl>Categoria1
  <dl><du>
    <dl>Categoria2
    <dl><du>
      <dl>Riferimento
    </du>
  </du>
</du>
```

Si noti come l'impostazione del testo HTML dei risultati sia definita in un file di template; la classe `Find` implementa un `Generator` sensibile alla stringa `[RESULT]`.

Per motivi statistici e di *testing*, la classe `Find` memorizza in un particolare direttorio (anch'esso configurabile in fase di installazione) la chiave di ricerca immessa dall'utente, prima di qualunque elaborazione sintattica. In particolare, nel direttorio viene creato un file che ha per nome la data e l'ora corrispondente ad ogni accesso al sistema. Non è prevista alcuna sincronizzazione sull'accesso a quest'altro direttorio, in quanto le informazioni ivi memorizzate non sono significative per il servizio (non è fondamentale cioè che siano tutte corrette), ma servono esclusivamente a monitorare il comportamento degli utilizzatori. È proprio in base a questo comportamento che sarà possibile decidere quali estensioni realizzare a questo progetto.

7.2.4 Invio domanda all'esperto

La classe `Mail` si occupa del recapito della domanda posta dall'utente presso l'esperto. Il suo funzionamento ricalca esattamente quanto descritto per il BLOCCO DI RISPOSTA al paragrafo 5.2.5; il costruttore della classe `Mail` è infatti definito nel modo seguente:

```
public Mail(Catalog c, String dirList, String title, String quest) {
  ...
}
```

in cui:

- `c` è la struttura della BASE DI CONOSCENZA;
- `dirList` è l'elenco dei direttori corrispondenti alle categorie selezionate dall'utente;
- `title` è il titolo della domanda;
- `quest` è il testo della domanda.

Come già spiegato, prima di scrivere sul direttorio QUEST la classe `Mail` fa uso dei metodi offerti dalla classe `Sync` per ottenerne il diritto.

Il nome del file effettivamente scritto sul disco rigido corrispondente alla domanda viene costruito unendo la stringa corrispondente alla data odierna ed il titolo della domanda. In realtà, dal titolo viene eliminata una serie di caratteri che possono indurre malfunzionamenti nel *file system*. In particolare la sostituzione avviene con l'istruzione:

```
name = vString.replace(name, " :&‘/\\*?|\"’<>", ’_’);
```

Tale file è un documento HTML (anch'esso prodotto tramite file di template) in cui è presente una *form* che richiama il BLOCCO DI RISPOSTA dell'esperto; in particolare, la form è di questo tipo:

```
<html>
  ...
  <form method=post action=http://www.eurotalete.com/servlet/zAnsw>
    <textarea name="answ" rows=20 cols=60></textarea>
    ...
  </form>
</html>
```

In maniera analoga a quanto accadeva con la classe `Find` a proposito del monitoraggio del comportamento degli utenti, la classe `Mail` memorizza in un altro particolare direttorio tutte le domande poste dagli utenti all'esperto. Esse possono essere utilizzate, ad esempio, per studiare un algoritmo di elaborazione sintattica della domanda, tale da estrarre da essa le parola chiave significative per la ricerca.

7.3 Servlet z

La classe `z` è la classe principale dell'applicazione, nel senso che è quella richiamata quando l'utente esegue una nuova ricerca o invia una nuova domanda all'esperto.

Essa ha una struttura estremamente semplice:

- nel metodo `init` si trova il codice che verrà eseguito una sola volta, al primo caricamento della Servlet in memoria; esso ha dunque il compito di caricare dai file sul disco rigido una volta per tutte il valore dei parametri di configurazione del sistema (`z.cfg`) e la struttura della BASE DI CONOSCENZA (`z.cat`)

- il metodo `doPost` viene invece richiamato quando l'utente invia alla Servlet una richiesta di tipo *post*. Tale metodo deve compiere le seguenti azioni:

  - discriminare quale tipo di azione è stata richiesta da parte dell'utente: estrazione di riferimenti dalla BASE DI CONOSCENZA o invio di una domanda all'esperto;
  - mediante il metodo `getParameter` della classe `Servlet` estrarre il valore dei campi della form che l'utente ha compilato e richiamare l'opportuna classe, `Find` o `Mail`.

7.4 Servlet `zAnsw`

La classe `zAnsw` è la classe-Servlet che realizza il BLOCCO DI RISPOSTA; essa viene richiamata quando l'esperto risponde ad un quesito.

Il suo funzionamento è stato dettagliatamente descritto nel paragrafo 5.2.7.

7.5 Programma `Update`

La classe `Update` è la classe principale del BLOCCO DI AGGIORNAMENTO. Il metodo `main` è richiamato quando l'amministratore del sistema esegue l'aggiornamento delle informazioni contenute nella BASE DI CONOSCENZA.

In tale modo di funzionamento, la classe `Update` implementa l'interfaccia `Action`. Per ogni categoria della BASE DI CONOSCENZA viene richiamato il metodo `onBoth`. Esso esegue le elaborazioni descritte nel paragrafo 5.2.3; questo è un estratto (semplificato) del suo codice:

```
dumpSites(c, sites);
report(sites, repNew);

if (
  (new BatchExec(cmpCmd, ! BatchExec.ThrowOnExit)).getResult() != 0
) {
  new BatchExec(cpCmd, this.outStream, this.outStream);
  spider(sites, dbDir);
  new BatchExec(swishCmd, this.outStream, this.outStream);
  new BatchExec(rmCmd, this.outStream, this.outStream);
  Sync.copy(index + Config.get("bakExt"), index);
}
```

dove:

- `dumpSites(...)` scrive un di un file di testo nel direttorio corrispondente alla categoria l'elenco dei documenti ad essa associati;
- `report(...)` esegue il programma di *spider* **webcp** in modalità "report": viene dunque prodotto un ulteriore file di testo , nel quale vengono elencati gli attributi di dimensione e data di ultimo aggiornamento per ogni documento da indicizzare;
- `new BatchExec(cmpCmd,...)` esegue la comparazione fra il rapporto generato al passo precedente (chiamando `report`) e quello generato

nell'ultima indicizzazione; se differiscono è necessario operare un nuovo recupero ed indicizzazione della categoria;

- `new BatchExec(cpCmd,...)` aggiorna il file di rapporto;
- `spider(sites, dbDir)` recupera il contenuto dei documenti elencati nel file `sites` all'interno del direttorio `dbDir`;
- `new BatchExec(swishCmd,...)` indicizza il contenuto del direttorio `dbDir`, quindi i nuovi documenti che compongono la categoria;
- `new BatchExec(rmCmd,...)` elimina il direttorio `dbDir`;
- `Sync.copy(...)` esegue la copia *sicura* dell'indice appena costruito su quello obsoleto.

Il programma che realizza il BLOCCO DI AGGIORNAMENTO, come descritto nel paragrafo 5.2.3, deve anche fornire un metodo per l'indicizzazione esclusiva del direttorio CSR; tale metodo si chiama `mail` e alla sua invocazione seguono le seguenti tre operazioni:

```
(new File(indexPath)).mkdirs();
new BatchExec(swishCmd, this.outStream, this.outStream);
Sync.copy(index + Config.get("bakExt"), index);
```

dove:

- `mkdirs()` crea, ove non già presente, il direttorio CSR; serve a non provocare un errore quando l'esperto sta rispondendo per la prima volta ad una domanda;
- `new BatchExec(swishCmd,...)` manda in esecuzione il programma di indicizzazione (**swish**) sul direttorio CSR;
- `Sync.copy(...)` effettua la copia *sicura* dell'indice prodotto al passo precedente su quello obsoleto.

7.6 Applet zzz

L'Applet `zzz` realizza con un sol programma il BLOCCO DI INGRESSO ed il BLOCCO DI USCITA: è cioè un'Applet che consente all'utente di esprimere la chiave di ricerca (o la domanda da inviare all'esperto), esegue la Servlet `z`, ne interpreta i risultati e li presenta all'utente.

La comparazione fra la soluzione basata sull'uso di JavaScript ed HTML con l'approccio basato su Applet evidenzia la superiorità di quest'ultimo; l'Applet è dotata infatti di una velocità molto maggiore, è notevolmente più portabile rispetto a JavaScript ed è un programma in grado di evolvere in base ad eventuali nuove specifiche che verranno poste in sede di miglioramento del programma.

Dal punto di vista dello sviluppo dell'applicazione oggetto della presente tesi, questa Applet rappresenta un significativo approccio alla realizzazione delle estensioni descritte al capitolo 8.

Da un punto di vista tecnico, è molto significativo constatare come sia stato possibile realizzare tale Applet senza duplicazione di codice: essa cioè basa il suo funzionamento sullo *stesso codice* con cui è realizzata la Servlet `z`. Ad esempio, la lettura del file `z.cat` e la creazione dell'albero corrispondente alla BASE DI CONOSCENZA viene realizzata *estendendo* (cioè ereditando da) le classi `Catalog` e `Category`; le nuove classi (`GCatalog` e `GCategory`) sono dotate di metodi e variabili d'istanza particolari, tali da consentir loro di essere correttamente visualizzate e gestite dall'interfaccia ad eventi dell'Applet, ma vengono inizializzate dallo *stesso* metodo `readFrom` della classe `Category`.

Come dimostrazione, infine, dell'effettiva non comune portabilità delle applicazioni Java, è importante sottolineare il fatto che codice dell'Applet sia interpretato ugualmente bene dalle varie versioni dei più noti *browser*: Netscape (3.01 per Windows e Solaris, 4.0 per Windows) ed Explorer (3.0 e 4.0 per Windows) nonchè, ovviamente, da HotJava (1.0 per Windows e Solaris).

Per una rappresentazione grafica dell'Applet `zzz` si veda la figura 7.2.

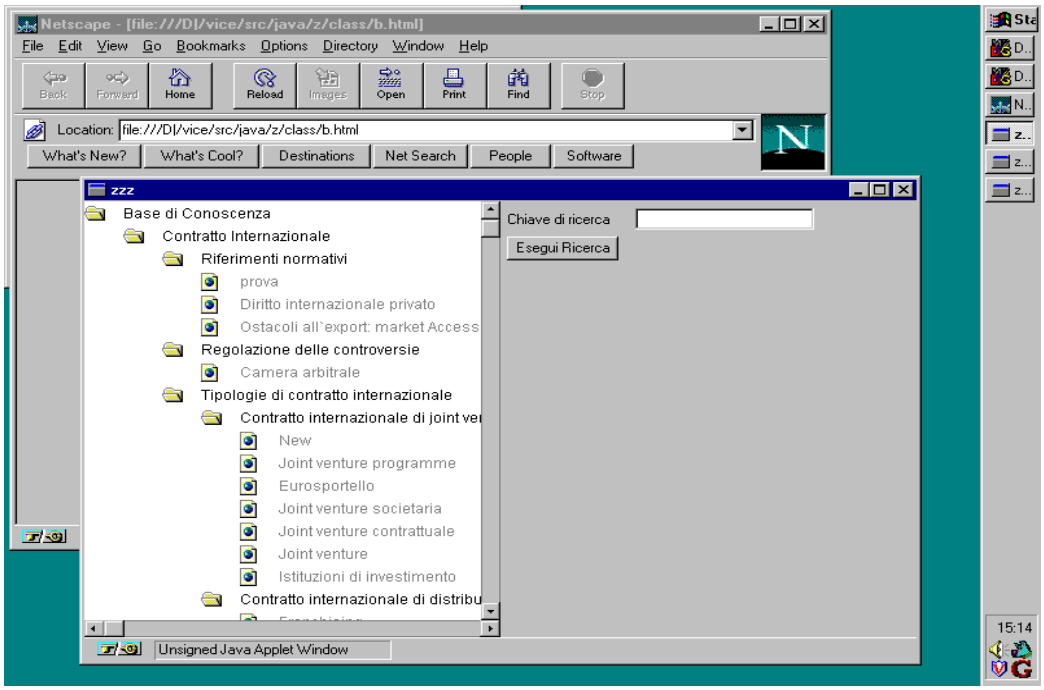

Figura 7.2: Applet zzz

Capitolo 8

Future estensioni

Il lavoro svolto e presentato in questa tesi rappresenta la prima versione del prototipo del sistema informativo descritto nel progetto comunitario Adapt.

In questo lavoro sono state analizzate e realizzate tutte le parti (blocchi funzionali) che compongono il programma, in maniera da soddisfare le specifiche poste in sede di progetto al capitolo 5.

Nel presente capitolo viene discusso come questo sistema potrà essere esteso e modificato per aumentarne le prestazioni, le capacità e l'utilità nei confronti dell'utente.

8.1 Modifiche architetturali

Applet

La prima modifica architetturale da esaminare riguarda la realizzazione di tutta l'interfaccia di ingresso ed uscita del programma mediante Applet, completando ed estendendo il lavoro iniziato con l'Applet `zzz` (vedere il paragrafo 7.6. Questo determinerà una maggiore velocità nell'interazione con l'utente ed una totale estendibilità del programma: qualunque nuova funzionalità sarà necessario realizzare, potrà essere ottenuta efficacemente mediante Applet.
In questo modo è ad esempio possibile fornire una soluzione al problema dell'aumento della struttura della BASE DI CONOSCENZA: sfruttando infatti il *multithreading* dell'ambiente Java potranno essere realizzati due *thread*: uno per la lettura della struttura delle categorie (il file `z.cat`) e la costruzione dell'albero, un secondo per la sua visualizzazione: in questo modo, il tempo che l'utente dovrebbe attendere prima di poter interagire con l'albero sarebbe costante (il tempo di caricamento dell'Applet in memoria) ed indipendente dalla dimensione della struttura delle categorie.

Connessioni con DataBase

Nel programma presentato in questa tesi la BASE DI DATI è formata esclusivamente da documenti HTML disponibili su Internet. Con l'obiettivo di fornire un servizio basato su una base di informazioni più vasta sarà necessario estendere il sistema informativo con la possibilità di eseguire ricerche su basi di dati (*DataBase*) già esistenti. Il BLOCCO DI RICERCA, in tal caso, non ricercherà più documenti indicizzati localmente alla macchina, ma dovrà interfacciarsi con il *DataBase*. Il modo in cui avverrà l'interrogazione, il tipo di informazioni ottenute ed i tempi di risposta dipenderanno totalmente dal particolare *DataBase* con cui ci si interfaccerà.

Modellizzazione degli utenti

La modifica architetturale concettualmente più importante dal punto di vista dell'Intelligenza Artificiale è la gestione di un modello d'utente[41], ovvero di un oggetto che raccolga le informazioni necessarie al sistema informativo per adattare (per quanto possibile) il servizio al tipo di utente. Il modello d'utente può essere impiegato ad esempio per riordinare in maniera differenziata i riferimenti estratti dalla BASE DI CONOSCENZA (si veda il paragrafo 8.3) o informare quell'utente di variazioni in documenti di suo particolare interesse; all'interno del modello d'utente può inoltre essere prevista la gestione dell'autentificazione degli utenti mediante controllo sull'accesso, al fine di impedire che utenti non autorizzati interroghino il sistema per il recupero delle informazioni o possano ottenere risposte dall'esperto.

Categorizzazione delle risposte

Le risposte dell'esperto vengono inserite in una categoria particolare della BASE DI CONOSCENZA (categoria CSR). Si può invece pensare di operare una *categorizzazione* delle risposte, cioè prevedere un loro inserimento nelle categorie corrispondenti all'argomento di cui trattano. L'indicazione della categoria in cui inserire la risposta dovrebbe essere fornita dall'esperto in linea.

È inoltre prevedibile lo sviluppo di un modulo per consentire all'esperto di *modificare* una risposta già categorizzata.

Configurazione automatica del sistema

La manutenzione della struttura delle categorie è manuale. È pensabile altresì la realizzazione di un modulo di programma in grado di generare auto-

maticamente `z.cat` dal file dei bookmarks di Netscape, la cui manutenzione risulterebbe estremamente semplificata.

8.2 Modifiche al BLOCCO DI INGRESSO

Le possibili modifiche al BLOCCO DI INGRESSO riguardano fondamentalmente la realizzazione di algoritmi più sofisticati per l'elaborazione della chiave di ricerca immessa dall'utente. In particolare, sono prevedibili le seguenti migliorie:

- selezione delle categorie con metodi alternativi alla selezione grafica sull'albero della BASE DI CONOSCENZA; ad esempio, si può pensare ad una selezione per sinonimia o similitudine;
- miglioramento dell'algoritmo mediante il quale vengono estratte le parole chiave significative per la ricerca; in particolare, si potrebbe realizzare un analizzatore sintattico che consentisse di unificare la richiesta di parole chiave per la ricerca con l'invio di una domanda all'esperto;
- inclusione nella chiave di ricerca dei sinonimi di ogni parola chiave digitata (o estratta dall'analizzatore sintattico), realizzando così una ricerca più generica.

Un altro tipo di modifiche riguarda l'interfaccia: quando sarà attivo il sistema per il riconoscimento degli utenti (si veda il paragrafo sull'autentificazione) sarà possibile sviluppare una sezione di "informazione" per l'utente, nella quale venga controllato, ad esempio:

- se l'esperto ha dato risposta ad alcune domande poste da quell'utente,
- se qualche documento particolarmente interessante per quell'utente è stato modificato.

8.3 Modifiche al BLOCCO DI USCITA

Le modifiche al BLOCCO DI USCITA riguardano invece algoritmi per presentare all'utente in maniera più sofisticata i riferimenti estratti:

- memorizzazione di una "anteprima" di ogni documento della BASE DI DATI. Col termine "anteprima" si intende un estratto significativo del documento, che possa essere presentato all'utente molto più rapidamente che non il suo intero contenuto; l'"anteprima" dovrebbe servire

all'utente per capire qual è il suo grado di interesse nei confronti di quel documento, in modo da reperirlo effettivamente solo se necessario;

- riordinamento complessivo dei riferimenti estratti in base alla loro corrispondenza con la chiave di ricerca. Si noti come **SWISH** esegua automaticamente un ordinamento dei riferimenti estratti da una medesima categoria; questo concetto può essere esteso realizzando un ordinamento reciproco anche fra riferimenti estratti da categorie distinte.
  SWISH calcola un valore di *ranking* per ogni riferimento estratto dall'indice, che viene poi normalizzato per assumere valori da 0 a 1000 prima di essere prodotto in uscita. Per ottenere una valutazione complessiva assoluta dei documenti (che dipenda cioè esclusivamente dalla chiave di ricerca e non dall'indice in cui si trovano) basta eliminare questa normalizzazione;

- estensione dell'algoritmo di ordinamento discusso al punto precedente con l'introduzione di un "modello d'utente"; il "modello d'utente" è un oggetto associato ad ogni utente che cosente al sistema informativo di avere informazioni sugli argomenti di interesse e di non interesse dell'utente[41].
  Per quanto riguarda il particolare sistema informativo oggetto di questa tesi, si può sfruttare la conoscenza semantica sulla strutturazione della BASE DI CONOSCENZA e riordinare *le categorie* (piuttosto che *i documenti*) in base al modello d'utente. Da un punto di vista del BLOCCO DI USCITA, questo comporterebbe una doppia possibilità di rappresentazione grafica del riferimenti estratti:

  - la rappresentazione "tradizionale", in cui i riferimenti ai documenti vengono visualizzati all'interno della porzione della struttura ad albero su cui è costruita la BASE DI CONOSCENZA;
  - ed una rappresentazione "tabulare", in cui i riferimenti ai documenti vengono visualizzati in una tabella (un riferimento per ogni riga) in cui i primi appartengono a categorie corrispondenti agli interessi dell'utente; gli ultimi a categorie corrispondenti ai non interessi dell'utente.

  L'interfaccia di uscita dovrà inoltre prevedere un metodo per consentire all'utente di specificare il suo grado di interesse verso un particolare documento: il sistema informativo potrà utilizzare tale specificazione (ed in particolare la conoscenza sulla categoria a cui quel documento appartiene) per modificare il modello di quell'utente.

Conclusioni

In questa tesi sono stati studiati gli strumenti che consentono ad un programma di interagire con la rete, unitamente ai sistemi per il recupero dell'informazione. In stretta collaborazione con il personale tecnico della società Talete è stata realizzata la prima versione del sistema informativo, presentato il 21 gennaio 1998 ad operatori del settore ed ora già in fase di prima distribuzione presso i possibili futuri utenti.

Il sistema, in questa fase prototipale, si compone di una interfaccia utente che prevede la possibilità di navigare attraverso le categorie, di interrogare mediante chiavi di ricerca la base di conoscenza e di inviare domande in linguaggio naturale ad esperti. Il sistema è in grado di estrarre riferimenti ai documenti che compongono la propria base di conoscenza; quest'ultima consta degli indici, creati in maniera automatizzata e memorizzati all'interno del sito, di documenti distribuiti sulla rete Internet e dei documenti appresi come risposte da parte dell'esperto.

Il sistema fornisce infine una interfaccia di uscita omogenea (con quella di ingresso) verso l'utente, fornendo come risposta ad una interrogazione la lista strutturata di tutti i documenti trovati.

I tempi di risposta sono pressochè immediati per interrogazioni all'interno della base di conoscenza (dipendono esclusivamente dalla banda del collegamento), mentre si richiede qualche giorno per l'elaborazione della risposta da parte dell'esperto umano.

L'evoluzione del sistema nei prossimi mesi sarà determinata sia dalle considerazioni svolte nel capitolo 8, che renderanno il servizio offerto complessivamente più sofisticato, sia dalle richieste e dai suggerimenti che proverranno direttamente dagli utilizzatori, in modo da rendere il servizio maggiormente adeguato alle loro esigenze.

Colgo l'occasione per ringraziare il personale tecnico delle socità Talete e Zuffellato Computers per la grande disponibilità e collaborazione offerte.

Bibliografia

[1] A. Gupta, R. Jain, *Visual Information Retrieval*, Communications of the ACM, maggio 1997 val 40, no. 5

[2] http://www.inktomi.com

[3] http://www.yahoo.com

[4] http://www.excite.com

[5] http://www.lycos.com

[6] http://www.webcrawler.com

[7] M. Koster, *The Web Robots FAQ...*
http://info.webcrawler.com/mak/projects/robots/faq.html

[8] http://www.altavista.com

[9] *Le FAQ del WebMaster di AltaVista*
http://altavista.telia.com/cgi-bin/query?mss=it/faq_webmaster&country=it

[10] http://www.hotbot.com

[11] http://www.metacrawler.com

[12] http://htdig.sdsu.edu

[13] http://www.altavista.com

[14] ftp://ftp.eit.com/pub/web.software/swish

[15] *Il Data Base dei Robots*
http://info.webcrawler.com/mak/projects/robots/active/html/index.html

[16] http://www.acme.com

[17] http://java.sun.com

[18] Batini, Carlucci Aiello, Spaccamela, Lenzerini, Marchetti, *Fondamenti di programmazione dei calcolatori elettronici*, Scienze e tecnologie informatiche, 1993

[19] Stefan Münz, *Dokumentation: JavaScript*, 1997
http://www.1ateam.com/1ateam/surfer/html/self/selfhtml.htm

[20] Bertrand Meyer, *La produzione del software object-oriented*, Prentice Hall International, 1991

[21] Hubert Pertl, *HTML Einführung*, Sep 1997,
http://www.boku.ac.at/htmleinf/

[22] Patrick Naughton, *Il manuale Java*, McGraw Hill, 1996

[23] Laura Lemay, *Java in 21 Tagen*, Markt&Technik, 1996,
http://germany.web.aol.com/mut/httoc.htm

[24] George Eckel, *Creare un server Internet con Unix*, Jackson Libri, 1996

[25] Brian W. Kernighan, Dennis M. Ritchie, *Il linguaggio C*, Jackson Libri

[26] K. E. Gorlen, S. M. Orlow, P. S. Plexico, *La programmazione ad oggetti e tipi di dati astratti in C++*, tecniche nuove, 1990

[27] Guy L. Steele, *Common Lisp The Language*, 2nd edition, Thinking Machines Inc., Digital Press, 1990

[28] David Ungar, *Generation Scavenging: a non-disruptive high performance storage reclamation algorithm*, in Proc. ACM SIGSOFT/SIGPLAN Software Engineering Symp. on Practical Software Developements Environments (Pittsburgh, Pennsylvania, apr. 1984), maggio 1984.

[29] Dijkstra, Lamport, Martin, Scholten, Steffens, *On-the-fly garbage collection: an excercise in cooperation*, Communications of the ACM, vol 21, no. 11, Nov. 1978

[30] Susan Dumais, *Finding what you want: new tools and tricks*, IEEE Software, Sep. 1995

[31] Daniel O'Leary, *AI and navigation on the Internet and Intranet*, IEEE Expert, Apr. 1996

[32] Daniel O'Leary, *The Internet, Internets, and the AI Renaissance*, IEEE Computer, Jan. 1997

[33] W. Bruce Croft, *Effective Text Retrieval based on the combining evidence from the corpus and users*, IEEE Expert, Dec. 1995

[34] Chris Drummond, Dan Ionescu, Robert Holte, Nicolas Georganas, Emil Petriu, *Intelligent browsing for multimedia applications*, IEEE Proceedings of MULTIMEDIA, 1996

[35] Beerud Sheth, Pattie Maes, *Evolving agents for personalized information filtering*, IEEE, 1993

[36] Michael Pazzani, Larry Nguyen, Stefanus Mantik, *Learning for hotlists and coldlists: toward a WWW information filtering and seeking agent*, IEEE, 1995

[37] M. F. Wyle, *A wide area network information filter*, IEEE, 1991

[38] Erik Selberg, Oren Etzioni, *The Metacrawler architecture for resource aggregation on the Web*, IEEE Expert, Jan-Feb. 1997

[39] Sougata Mukhrjea, Kyoji Hirata, Yoshinori Hara, *Visualizing the results of multimedia Web search engines*, IEEE, 1996

[40] K. Hammond, R. Burke, C. Martin, S. Lytinen, *FAQ Finder: a CASE based approach to knoledge navigation*, IEEE, 1995

[41] Marco Minio, Carlo Tasso, *IFT: un'interfaccia intelligente per il filtraggio di informazioni basato su modellizzazione d'utente*, AI*IA Notizie, Sep. 1996

[42] Kuhanandha Mahalingam, Michael N. Huhns, *A tool for organizing Web information*, IEEE Computer, Jun. 1997

[43] Carlo Tasso, *Il progetto FIRE*

[44] Fabio Asnicar, Massimo Di Fant, Carlo Tasso, *User model-based information filtering*

[45] Hendrik Eriksson, *Expert System as knowledge servers*, IEEE Expert, 1996

Appendice A

Configurazione del sistema

Di seguito verranno descritti i file necessari per la configurazione del sistema informativo su una particolare macchina. Il loro contenuto effettivamente presentato in questa appendice è quello impiegato per il funzionamento sul sito di Talete `www.eurotalete.com`.

A.1 `z.cfg`

In questa appendice viene presentato il file `z.cfg`, che contiene l'elenco dei parametri configurabili del sistema informativo.

Il significato dei parametri è in genere autoesplicativo. Si noti altresì come venga utilizzato il carattere `/` come separatore dei nomi dei file e non \ come tradizionalmente valido in ambiente Windows; questa particolarità è già stata discussa nel paragrafo 6.1.4.

```
#
# configurazione per www.eurotalete.com
#

#
# sintassi di z.cat
#
SiteOpen          "sitebegin"
SiteClose         "siteend"
CatOpen           "catbegin"
CatClose          "catend"

#
# template
```

```
#
HTMLForm          "/zuff/Tmpl/iform.htm"
HTMLSearch        "/zuff/Tmpl/motore.htm"
HTMLAnsw          "/zuff/Tmpl/answ.htm"
HTMLNavig         "/zuff/Tmpl/servizi.htm"
HTMLQuest         "/zuff/Tmpl/quest.htm"
HTMLSent          "/zuff/Tmpl/sent.htm"
HTMLResult        "/zuff/Tmpl/result.htm"
HTMLCSRUpdate     "/zuff/Tmpl/csrupdate.htm"
HTMLIndex         "/zuff/Tmpl/index.htm"

#
# config. Update
#
Extensions        ".html .htm .txt"
SitesName         "sites.txt"
ReportNew         "report.new"
Report            "report.txt"
cmpCmd            "d:/vice/bin/cmp.exe -s "
cpCmd             "d:/vice/bin/cp.exe "
idxCmd "d:/zuff/exe/swish.exe -c /zuff/config/swish.cfg -i "
bakExt            ".bak"
rmCmd             "d:/vice/bin/rm.exe -Rf "

#
# config. Find
#
searchCmd         "d:/zuff/exe/swish.exe"
dbIndexDir        "d:/db"
FolderImg "http://www.eurotalete.com/adapt/images/closedfolder.gif"
DocImg            "http://www.eurotalete.com/adapt/images/doc.gif"

#
# directories
#
questDir          "d:/zuff/quest"
questBakDir       "d:/zuff/questBak"
dbDir             "d:/db"
rootDir           "d:/zuff"
CSRDir            "d:/zuff/csr"
CSRIndex          "d:/zuff/csr/index.htm"
```

```
searchBakDir      "d:/zuff/searchBak"

#
# generale
#
Catalog           "z.cat"
MailLog           "/zuff/quest/mail.txt"
MailHRef          "http://www.eurotalete.com/quest/mail.txt"
Separator         "/"
Semaphore         "_semaphore"
Index             "index.swish"
CSRIdx            "csr"
```

A.2 z.cat

Il file `z.cat` mantiene la descrizione della BASE DI CONOSCENZA. La sintassi con cui le categorie vengono dichiarate è definita dai parametri `CatOpen`, `CatClose`, `SiteOpen`, `SiteClose` del file `z.cfg`. Ogni riga contenente l'indicazione di un sito ha la seguente sintassi:

$$[\text{"}etichetta\text{"}]\ [+n]url$$

dove:

- *etichetta* è l'eventuale stringa alfanumerica con quel documento verrà nominato all'interno del programma;
- $+n$ indica la distanza (misurata in numero di riferimenti ipertestuali) da percorrere nel recupero dei documenti correlati a quello corrente; n è una cifra decimale (possono essere recuperati documenti distanti fino a 9 riferimenti ipertestuali) o `'+'` per non porre limiti a tale distanza;
- *url* è la URL del documento.

Uno stralcio del file `z.cat` effettivamente impiegato come BASE DI CONOSCENZA è il seguente:

```
catbegin "Base di Conoscenza" zk
  catbegin "Contratto Internazionale" conint
    catbegin "riferimenti normativi" rifnor
      sitebegin
      "diritto internazionale privato"
```

```
        http://www.italia.informest.it/ital/c8-7-3.htm
      "ostacoli all'export: market Access Database"
        +2http://mkaccdb.eu.int/
      siteend
    catend
    catbegin "regolazione delle controversie" regcnv
      sitebegin
      "camera arbitrale" http://www.mi.camcom.it/camera.arbitrale/
      siteend
    catend
  ...
  ...
  catbegin "case-studies risolti dagli esperti" csr
    sitebegin
    "Indice" http://www.eurotalete.com/csr/index.html
    siteend
  catend
catend
```

A.3 swish.cfg

Il file `swish.cfg` è il file di configurazione per **SWISH** in fase di indicizzazione. La parte più significativa è rappresentata dalle linee con `ReplaceRules`. Esse sono necessarie per memorizzare all'interno dell'indice prodotto le URL dei documenti indicizzati, sostituendo cioè il nome del direttorio in cui si trovano i documenti da indicizzare (corrispondente al parametro `dbDir`) con la stringa `http:/`; se ad esempio si dovesse indicizzare il documento

```
d:/db/www.server.it/direttorio/documento.html
```

nell'indice verrebbe memorizzata la corrispondente URL

```
http://www.server.it/direttorio/documento.html
```

```
#
# swish.cfg
#
IndexReport 3
IndexOnly .html .htm .txt .java .c .h
ReplaceRules replace "e:/db" "http:/"
ReplaceRules replace "d:/db" "http:/"
```

```
ReplaceRules replace "d:/zuff/csr" "http://www.eurotalete.com"
IgnoreLimit 50 100
IgnoreWords SwishDefault
```

Appendice B

Sorgenti

In questa sezione viene riportato uno stralcio della parte più significativa dei sorgenti del programma.

```
//
// zAnsw.java
//
// Andrea Vicentini 05/01/1998
//

import vice.*;
import zuf.*;
import java.io.*;
import java.util.NoSuchElementException;

import javax.servlet.*;
import javax.servlet.http.*;


public class zAnsw4 extends HttpServlet implements Generator {

  protected static final String DIRSTR = "[DIR]";
  protected static final String ConfigStr = "/zuff/config/z.cfg";

  protected Catalog c;
```

```
  public static void main(String args[]) {
    (new zAnsw4()).go_cgi_normal(ConfigStr);
  }

  protected void go_cgi_normal(String cfg) {
    try {
      Config.setUp(ConfigStr);
      c = new Catalog(Config.get("Catalog"));

      Cgi.printHdr();
      if (Cgi.thereIsInput()) {

        try {
          MimeDec d =
            new MimeDec((new DataInputStream(System.in)).readLine());

          nameP = d.get("name");
          questP = d.get("quest");
          titleP = d.get("title");
          catP = d.get("cat");
          answP = d.get("answ");
          dateP = d.get("date");

          go();
        } catch (IOException e) {
          System.out.println("<html>I/O error: " +
          e.getMessage() + "</html>");
        }
      } else
        System.out.println(
"<html>Non e' cosi' che mi eseguirai!</html>");

    } catch (ViceExitError e) {
      System.out.println(
        "<html>Si e' verificato un errore:<hr>" +
        e.getMessage() + "<hr></html>"
      );
    }
  }

  protected String nameP;
```

```
  protected String questP;
  protected String titleP;
  protected String catP;
  protected String answP;
  protected String dateP;


  public void doPost(HttpServletRequest req, HttpServletResponse res)
  throws ServletException, IOException {

      // grazie jdk1.1.x!
    System.setOut( new PrintStream( res.getOutputStream() ) );

    try {
      Config.setUp(ConfigStr);
      c = new Catalog(Config.get("Catalog"));

      try {
        nameP = req.getParameter("name");
        questP = req.getParameter("quest");
        titleP = req.getParameter("title");
        catP = req.getParameter("cat");
        answP = req.getParameter("answ");
        dateP = req.getParameter("date");

        go();
      } catch (IOException e) {
        throw new ViceExitError("(zAnsw): " + e.getMessage());
      }

    } catch (ViceExitError e) {
      System.out.println(
        "<html>Si e' verificato un errore:<hr>" +
        e.getMessage() + "<hr></html>"
      );
    }
  }


  protected void go() throws IOException {
    String answStr =
```

```
      Config.get("CSRDir") + Config.get("Separator") + nameP.trim();

    String questStr =
      Config.get("questDir") + Config.get("Separator") + nameP.trim();


    fromQuestToAnsw(
      questStr, answStr,
      dateP, titleP, catP, questP, answP
    );
    listRebuild(Config.get("HTMLIndex"), Config.get("CSRIndex"));
    updateCsr(Config.get("MailLog"));

      // cancello la domanda
    String error;
    if ( ! (new File(questStr)).delete() )
      error = "Non posso cancellare " + questStr;
    else
      error = "La domanda e' stata correttamente cancellata.";

    warnOut(error);
  }


  protected void fromQuestToAnsw(
    String questStr, String answStr,
    String dateP, String titleP, String catP,
    String questP, String answP
  ) throws IOException {
    File questFile = new File(questStr);

    Templater t = new Templater(Config.get("HTMLAnsw"));
    t.put(new SimpleGen("[DATE]", dateP));
    t.put(new SimpleGen("[TITLE]", titleP));
    t.put(new GenCat(c, catP));
    t.put(new SimpleGen("[QUEST]", questP));
    t.put(new SimpleGen("[ANSW]", answP));

    (new File(Config.get("CSRDir"))).mkdirs();

    if ( (new File(answStr)).exists() )
```

```
      throw new ViceExitError(
        "(zAnsw): " + answStr + " esiste gia'...<br>" +
        "Qualcosa deve essere andato storto."
      );

      // scrivo il file di risposta
    FileOutputStream answFile = new FileOutputStream(answStr);
    t.go(new PrintStream(answFile));
    answFile.close();
  }



  protected void updateCsr(String logStr) throws IOException {
    FileOutputStream outStreamTmp = new FileOutputStream(logStr);
    PrintStream outStream = new PrintStream(outStreamTmp);

    try {
      c.find(Config.get("CSRIdx"));
      String indexPath =
        ((Category) ((Node) c.Current() ).getObj()).getPath();
      String index =
        indexPath + Config.get("Separator") + Config.get("Index");
      String swishCmd =
        Config.get("idxCmd") + " " + Config.get("CSRDir") +
        " -f " + index + Config.get("bakExt");

      (new File(indexPath)).mkdirs();

      new BatchExec(swishCmd, outStream, outStream);

      Sync.copy(index + Config.get("bakExt"), index);

    } catch (NoSuchElementException e) {
      throw new ViceExitError(
        "(Update -- mail): non c'e' la categoria di nome " +
        Config.get("CSRIdx")
      );

    } catch (IOException e) {
      throw new IOException("(Update -- mail): " + e.getMessage());
```

```
    }

    outStreamTmp.close();
  }


  protected void listRebuild(String source, String dest)
  throws IOException {
    Templater t = new Templater(source);
    FileOutputStream outStreamTmp =
      new FileOutputStream(dest + Config.get("bakExt"));
    PrintStream outStream = new PrintStream(outStreamTmp);

    t.put(this);
    t.go(outStream);
    outStreamTmp.close();
    Sync.copy(dest + Config.get("bakExt"), dest);
  }


  public boolean matches(String s)
    { return (s.indexOf(DIRSTR) != -1); }
  public String explode() {
    StringBuffer result = new StringBuffer();
    String index = Config.get("CSRIndex");
    String indexBak = Config.get("CSRIndex") + Config.get("bakExt");
    String date, title;
    int DateLen = 8;
    int HtmlLen = 5;

      // ordinamento dal + recente
    String [] ls = sort( (new File(Config.get("CSRDir"))).list() );

    OTree idxT = new OTree("Indice delle risposte:");

    for (int i = 0; i < ls.length; i++)
     if (!index.endsWith(ls[i]) && !indexBak.endsWith(ls[i])) {

          // prelevo la data ed il titolo
        date = ls[i].substring(0, DateLen);
        title =
```

```
  ls[i].substring(DateLen + 1, ls[i].length() - HtmlLen);

        idxT.mkBranch(date, " ");
        ( (OCategory) ( (Node) idxT.Current() ).getObj() ).addLine(
          "<rank> " + ls[i] + " \"" + title + "\" <dim>"
        );
      }

/*
  ### plain output
        result.append(
          "<a href=\"" +
          Config.get("CSRUrl") + ls[i] + "\">" +
          ls[i].substring(0, ls[i].length() - 5) + "</a><br>"
        );
    return result.toString();
*/
    return idxT.htmllist();

  }

  protected void warnOut(String error) throws IOException {
    Templater t = new Templater(Config.get("HTMLCSRUpdate"));
    t.put(new SimpleGen("[ERROR]", error));
    t.put(new SimpleGen("[LOG]", Config.get("MailLog")));
    t.put(new SimpleGen("[HREF]", Config.get("MailHRef")));
    t.go();
  }


  protected String[] sort( String [] v ) {
    int c1, c2;
    int max;

    for (c1 = 0; c1 < v.length - 1; c1++) {

        // ricerca maggiore in c1 + 1 .. length
      max = c1;
      for (c2 = c1 + 1; c2 < v.length; c2++)
        if (compare(v[max], v[c2]) < 0)
          max = c2;
```

```
        // swap if necessary
      String tmp = v[c1];
      v[c1] = v[max];
      v[max] = tmp;
    }

    return v;
  }

  protected int compare(String s1, String s2) {
    return dateStr(s1).compareTo(dateStr(s2));
  }

      // 13-01-98 ==> 980113
  protected String dateStr(String s) {
    return s.substring(6, 8) + s.substring(3, 5) + s.substring(0, 2);
  }
}



//
// Linkable.java
//
// Andrea Vicentini 14/01/1998
//

package vice;


////////////
//
//  Gli implementatori di questa interfaccia saranno
//  linkabili come oggetti all'interno di un albero.
//
public interface Linkable {

////////////
//
//  Ritorna il nodo corrispondente a questo elemento.
```

```
//
  public Node getFather();

////////////
//
//  Setta il valore della chiave di questo elemento
//  indicizzabile.
//
  public void setKey(Object s);

////////////
//
//  Controlla se <code>key</code> corrisponde a questo oggetto
//  indicizzato.
//
  public boolean matches(Object key);
}



//
// MkNode.java
//
// Andrea Vicentini 04/01/1998
//

package vice;


import java.util.*;


public class MkNode extends Node {

  public MkNode(TreeProducer p) {
    super(p);
  }

////////////
//
//  Cerca tra i suoi figli quello che si chiama come indicato,
```

```
//  aggiungendolo se non lo trova.
//  @return il nuovo figlio.
//
  public Node mkChild(Object key) {
    boolean found = false;
    Node result = getChild();
    Node status = null;

//System.out.println("*** vice.mkChild -- " + key);
    while ( (result != null) && ! found )
      if ( result.getObj().matches(key) )
        found = true;
      else {
        status = result;
        result = result.getBrother();
      }

    if (! found) {
      Linkable newObj = producer.newObj(this);
      newObj.setKey(key);

      if (status == null) {
//System.out.println("*** new child " + key);
        child = producer.newNode();
        child.obj = newObj;
        return child;

      } else {
        status.brother = producer.newNode();
        status.getBrother().obj = newObj;
        return status.getBrother();
      }

    } else
      return result;
  }


////////////
//
//  Costruisce un percorso a partire dal nodo corrente.
```

```
//  <tt>s</tt> e' un elenco di label di categorie, separate da
//  blank o newline.<br>
//  <b>NB:</b> il primo elemento e' saltato.
//
  public Node mkBranch(String s, String sep) {
    String newLabel;
    MkNode result, tmp;

    result = this;

    StringTokenizer inStream = new StringTokenizer(s, sep);
//    inStream.nextToken();
    while (inStream.hasMoreTokens()) {
      result = (MkNode) result.mkChild(inStream.nextToken());
    }

//    current = result;
    return result;
  }
}



//
// RdNode.java
//
// Andrea Vicentini 04/01/1998
//

package vice;

import java.io.*;


/////////
//
//  Gli elementi di questo tipo realizzano un nodo di un
//  albero caricabile da disco.
//  @see vice.TreeProducer, vice.MkNode, vice.RdNode,
//  vice.Keyable, vice.Readable
//
```

```
public class RdNode extends Node {

  protected TreeRdProducer producer;


  public RdNode(TreeRdProducer p) {
    super(p);
    producer = p;
  }


/////////
//
//  Legge dallo stream indicato il valore del nodo corrente.
//  @exception java.io.IOException in caso di
//  malfunzionamento.
//
  public void readFrom(AssertTokenizer inStream)
  throws IOException {
    String open = producer.getOpenStr();
    String close = producer.getCloseStr();

    obj = producer.newObj(this);
    ((Readable)obj).readFrom(
      inStream, producer.getOpenStr(), producer.getCloseStr()
    );

    inStream.next();
    if (inStream.sval.equals(open)) {
      inStream.pushBack();
      child = producer.newNode();
      ((RdNode)child).readFrom(inStream);

        // don't allow sites/snm after child category...
      inStream.next(close);
    }

    if (inStream.sval.equals(close)) {
        // ora controllo se e' la fine del file bzw ho fratelli
      inStream.nextToken();
      if (inStream.ttype == AssertTokenizer.TT_WORD) {
```

```
        inStream.pushBack();

        if (inStream.sval.equals(open)) {
          brother = producer.newNode();
          ((RdNode)brother).readFrom(inStream);
        }
      }
    }
  }
}



//
// Readable.java
//
// Andrea Vicentini 04/01/1998
//

package vice;

import java.io.*;


/////////
//
//  Le classi che implementeranno questa interfaccia potranno
//  caricare i loro valori da disco.
//  @see vice.AssertTokenizer, vice.RdNode
//
public interface Readable {

/////////
//
//  Legge da disco.
//  @param inStream stream da cui leggere i valori
//  @param open delimitatore di inizio dei valori per la
//  classe corrente
//  @param close delimitatore del termine della classe
//  corrente. Quando da inStream sara' letto il valore close
//  bisognera' terminare le operazioni di lettura
```

```
//
  public void readFrom(
    AssertTokenizer inStream, String open, String close
  ) throws IOException;
}



//
// Sync.java
//
// Andrea Vicentini 05/01/1998
//
// .980107. reset()
//

package vice;

import java.io.*;


/////////
//
//  Questa classe consente di eseguire operazioni
//  sincronizzate fra diversi processi, risolvendo eventuali
//  problemi di condivisione di risorse.
//
public class Sync {

/////////
//
//  Nome del file usato come semaforo.
//
  protected String name;

/////////
//
//  Path del file usato come semaforo.
//
  protected String path;
```

```
/////////
//
//  Inizializza le variabili e si sincronizza sul semaforo
//  (file) indicato.
//  @exception java.io.IOException in caso di malfunzionamenti
//
  public Sync(String path, String name) throws IOException {
    this.path = path;
    this.name = name;
    waitOn();
  }


/////////
//
//  Esegue effettivamente la sincronizzazione.
//  Sostanzialmente, si mette in attesa al <it>semaforo</it>:
//  <tt>waitOn</tt> termina quando il processo che occupava la
//  risorsa associata al semaforo la rilascia: il semaforo si
//  libera e consente al primo processo in attesa di entrare in
//  esecuzione.
//
  protected void waitOn() throws IOException {
    (new File(path)).mkdirs();
    File f = new File(path, name);
//System.out.println("waiting on " + path + name + " ...");
    while (!f.delete());
//System.out.println("go!");
  }


/////////
//
//  Ripristina il semaforo.
//  Serve ad evitare che, a causa di un malfunzionamento del
//  disco, il semaforo non venga piu' ripristinato, causando
//  l'attesa in coda di tutti i processi che necessitano della
//  risorsa associata. Dovrebbe essere chiamato da un blocco
//  <tt>finally</tt><br>
//  In caso di malfunzionamento genera un
```

```
//  <tt>ViceExitError</tt>, causando il termine del programma
//  con messaggio d'errore.
//
  public void reset() {
    try {
      release();
    } catch (IOException e) {
      throw new ViceExitError(
        "Non posso ricreare il semaforo " + path + File.separator +
        name + "\n" + e.getMessage()
      );
    }
  }


/////////
//
//  Rilascia la risorsa associata al semaforo ripristinandolo.
//  Rispetto a <tt>reset</tt> non assicura di ripristinare il
//  semaforo, in quanto puo' terminare anche solo semplicemente
//  con una IOExcpeption.
//  @exception java.io.IOException in caso di malfunzionamenti
//
  public void release() throws IOException {
    FileOutputStream outStream =
      new FileOutputStream(path + File.separator + name);
    outStream.close();
  }


/////////
//
//  Esegue una copia sincronizzata dal file associato a
//  sourceStr al file associato a destStr.
//  La sincronizzazione consiste nell'attesa che i processi
//  concorrenti rilascino il file associato a destStr. Quando
//  nessuno lo sta usando, la copia ha luogo.
//
  public static void copy(String sourceStr, String destStr)
  throws IOException {
    File f;
```

```
      // sincronizzazione
    f = new File(destStr);
    if (f.exists())
      while (!f.delete())
        ;

    if ( ! (new File(sourceStr)).renameTo(f) )
      throw new IOException(
        "Copia sincronizzata di " + sourceStr + " in " +
        destStr + " fallita"
      );
  }
}



//
// TreeProducer.java
//
// Andrea Vicentini 30/12/1997
//
// .980104. StateMachine
//


package vice;


/////////
//
//  Le classi che implementano questa interfaccia sono in grado
//  di generare un determinato tipo di albero.
//  @see vice.TreeRdProducer, vice.Node
//
public interface TreeProducer extends StateMachine {

/////////
//
//  Metodo che ritorna un riferimento ad un nuovo elemento
//  contenuto all'interno del nodo dell'albero.
```

```
//
  public Linkable newObj(Node n);

/////////
//
//  Metodo che ritorna un riferimento ad un nuovo nodo
//  dell'albero.
//
  public Node newNode();
}



//
// TreeRdProducer.java
//
// andrea Vicentini 04/01/1998
//

package vice;


/////////
//
//  E' una interfaccia che consente, agli alberi leggibi da
//  disco, di configurarsi in base alle parole che corrispondono
//  all'inizio ed alla fine di ogni nodo.
//  @see vice.Readable, vice.RdNode
//
public interface TreeRdProducer extends TreeProducer {

/////////
//
//  Ritorna la stringa corrispondente all'inizio di un nodo.
//
  public String getOpenStr();

/////////
//
//  Ritorna la stringa corrispondente alla fine di un nodo.
//
```

```
  public String getCloseStr();
}



//
// Catalog.java
//
// Andrea Vicentini 09/11/1997
//
// .971126. traccia dei livelli
// .971128. attraversamento dell'albero in modo infisso
// .971213. stampa labelsOf per l'output
// .971214. javadoc
// .980105. labelsOf elimina rootDir
//

package zuf;

import vice.*;
import java.io.*;
import java.util.*;



////////////
//
//  Mantiene la struttura ad albero di tutte le categorie.
//
public class Catalog implements TreeRdProducer {

///////////
//
//  Rende visibile la radice dell'albero.
//
  public RdNode getRoot() { return root; }
  protected RdNode root;

  protected AssertTokenizer inStream;

  protected Object current;
```

```
  public Object Current() { return current; }


  public void find(Object key) throws NoSuchElementException {
    if (
      (root == null) ||
      ((current = root.find(key)) == null)
    )
      throw new NoSuchElementException(key + " not found");
  }


  public Catalog(InputStream is) {
    try {
      inStream = new AssertTokenizer(is);
      go();
    } catch (IOException e) {
      throw new ViceExitError(
        "Errore nella lettura del catalogo.<br>" + e.getMessage()
      );
    }
  }


  public Catalog(String s) {
    InputStream inStreamTmp;

    try {
      inStreamTmp = new FileInputStream(s);
      inStream = new AssertTokenizer(inStreamTmp);
      go();
      inStreamTmp.close();

    } catch (FileNotFoundException e) {
      throw new ViceExitError(
"(zuf.Catalog): Non trovo il catalogo " + s
      );

    } catch (IOException e) {
      throw new ViceExitError(
        "Errore nella lettura del catalogo.<br>" + e.getMessage()
```

```
      );

    } catch (ViceExitError e) {
      throw new ViceExitError(
        "Errore nella lettura del catalogo.<br>" + e.getMessage()
      );
    }
  }


//////////
//
//  Apre il file indicato e carica in memoria l'albero.
//  <br>La sintassi con cui l'abero e' scritto e' configurabile.
//  <br>Nel caso di errori di sintassi termina.
//
  protected void go() throws IOException {
    inStream.eolIsSignificant(false);
    inStream.whitespaceChars(' ', ' ');  // portab. ?
    inStream.wordChars('$', '~');
    inStream.commentChar('#');
    inStream.quoteChar(Category.QUOTE);

    root = new RdNode(this);
    root.readFrom(inStream);
    setUpTree(root, Config.get("rootDir"), 0);
    current = root;
  }


  public Node newNode() { return new RdNode(this); }
  public Linkable newObj(Node n) { return new Category(n); }
  public String getOpenStr() { return Config.get("CatOpen"); }
  public String getCloseStr() { return Config.get("CatClose"); }



//////////
//
//  Stampa l'elenco delle label delle categorie che compongono
//  il percorso per raggiungere la categoria indicata.
```

```
//  Sono separate da <tt>\n</tt>.
//  La root-label non e' stampata.
//
  public String labelsOf(String catId) {
    return labelsOf(catId, "\n");
  }


//
//  Stampa l'elenco delle label delle categorie che compongono
//  il percorso per raggiungere la categoria indicata.
//  Sono separate da <tt>sep</tt>.
//  Viene eliminata la parte di percorso iniziale
//  (<code>rootDir</code>).
//  Produce una variazione nello <it>stato</it> dell'albero.
//
  public String labelsOf(String catId, String sep) {
    StringBuffer result = new StringBuffer();
    StringTokenizer inStream =
      new StringTokenizer(
        catId.substring(
          Config.get("rootDir").length()
        ).replace('/', ' ')
      );

      // salta il primo token (zk)
    inStream.nextToken();

    current = root;
    try {
      while (inStream.hasMoreTokens()) {
        current = ((Node) current).getChild( inStream.nextToken() );
        result.append(
          ( (Category) ((Node) current).getObj() ).getLabel() + sep
        );
      }
    } catch (NoSuchElementException e) {
      throw new ViceExitError(
        "(zuf.Catalog): Errore interno.<br>" + e.getMessage()
      );
    }
```

```
    String tmp = result.toString();
    if (tmp.endsWith(sep))
      return tmp = tmp.substring(0, tmp.length() - sep.length());

    if (! tmp.equals(""))
      return tmp;
    else
      return ((Category) getRoot().getObj() ).getLabel();
  }


//////////
//
//  Esegue <tt>setUp</tt> su tutti i nodi dell'albero.
//
  protected void setUpTree(Node t, String s, int level) {
    Category c = (Category) t.getObj();

    c.setUp(s, level);

    if (t.getChild() != null)
      setUpTree(t.getChild(), c.getPath(), level + 1);

    if (t.getBrother() != null)
      setUpTree(t.getBrother(), s, level);
  }
}



//
// Category.java
//
// Andrea Vicentini 09/11/1997
//
// .971126. traccia dei livelli nell'albero
// .971128. distinzione name / label
// .971202. sinonimi
// .971206. liberta' nella definizione delle categorie
// .971212. javadoc
```

```
// .971213. getChild per l'output
// .971214. extends Tree
// .980104. Keyable
//

package zuf;

import vice.*;
import java.util.Enumeration;
import java.util.NoSuchElementException;
import java.io.*;



//////////
//
//  Oggetto che mantiene le informazioni su una categoria
//
public class Category implements Readable, Linkable {

  public Node getFather() { return father; }
  public Object getKey() { return name; }
  public void setKey(Object s) { name = (String)s; }
  public boolean matches(Object s)
    { return name.equals( (String) s); }

//////////
//
//  Simbolo che delimita una stringa che puo' contenere spazi
//  nel file si disco.
//
  final public static int QUOTE = '"';

  protected Node father;

  protected String label;
  protected String dir;
  protected String name;
  protected int level;
```

```
//////////
//
//  Nome simbolico della categoria.
//
  public String getLabel() { return label; }

  public int getLevel() { return level; }

//////////
//
//  Path in cui la categoria si trova sul disco.
//  E' usato anche come identificatore univoco di categoria.
//
  public String getPath()  { return dir; }


    // eventuali siti della categoria con i sinonimi
  protected vVector snm;
  protected vVector sites;
  protected vVector files;


//////////
//
//  Rende disponibile l'elenco dei siti di questa categoria.
//
  public Enumeration getSites() { return sites.elements(); }


//////////
//
//  Rende disponibile l'elenco dei sinonimi di questa categoria.
//
  public Enumeration getSnm()   { return snm.elements(); }

  public Category(Node n)  { father = n; }


//////////
//
//  Costruisce una nuova categoria.
```

```
//  La chiamata e' ricorsiva.
//  @param inStream file da cui leggo i valori da inserire
//    nella categoria
//  @param path directory del nodo padre.
//    Sostanzialmente, e' la posizione nel file system sotto
//    cui si trovera' la categoria
//  @param lvl livello corrispondente a questa categoria
//  @exception java.io.IOException nel caso di problemi di
//    lettura da <tt>inStream</tt>
//
  public void readFrom(
    AssertTokenizer inStream, String open, String close
  ) throws IOException {

      // mi aspetto l'apertura di una nuova categoria
    inStream.next(open);

      // carica il nome e compone il path della directory
      // gli unici due elementi che sono fissi ed in quest'ordine
    inStream.next(QUOTE);
    label = inStream.sval;

    inStream.next();
    name = inStream.sval;

      // pulizia delle variabili...
    sites = new vVector();
    snm = new vVector();
    files = new vVector();

    inStream.next();
    while (
      ! inStream.sval.equals(close) && ! inStream.sval.equals(open)
    ) {

      if (inStream.sval.equals(Config.get("SinOpen")))
        snm.merge(inStream.loadVect(Config.get("SinClose")));

      else if (inStream.sval.equals(Config.get("SiteOpen")))
        readSites(sites, inStream, Config.get("SiteClose"));
```

```
      else if (inStream.sval.equals(Config.get("FileOpen")))
        files.merge(inStream.loadVect(Config.get("FileClose")));

      inStream.next();
    }

    inStream.pushBack();
  }


//////////
//
//  Sistema <tt>level</tt> e <tt>dir</tt> di questa categoria.
//
  public void setUp(String rootPath, int l) {
    dir = rootPath + Config.get("Separator") + name;
    level = l;
  }


//////////
//
//  Legge l'elenco dei siti.
//  Per ogni sito memorizza etichetta, profondita' di recupero e url.
//  @param s vettore risultante dei siti
//  @param inStream file da cui leggere
//  @param close stringa che delimita la fine dei siti
//
  protected void readSites(
    vVector s, AssertTokenizer inStream, String close
  ) throws IOException {

    inStream.nextToken();

    while ( !
      (
        (inStream.ttype == inStream.TT_WORD) &&
        (inStream.sval.equals(close))
      )
    ) {
      Site tmp = new Site();
```

```
      if (inStream.ttype == QUOTE) {
        tmp.label = inStream.sval;
        inStream.nextToken();
      }

      if (inStream.ttype == inStream.TT_WORD) {
        int c = inStream.sval.charAt(0);

        if (c == '+') {
          c = inStream.sval.charAt(1);
          if (c == '+')
            tmp.linkDepth = -1;
          else
            tmp.linkDepth = c - '0';

          tmp.url = inStream.sval.substring(2);
        } else
          tmp.url = inStream.sval;

      } else
        throw new IOException
  ("not word in line " + inStream.lineno());

      s.addElement(tmp);
      inStream.nextToken();
    }
  }
}



//
// Config.java
//
// Andrea Vicentini 17/11/1997
//
// .971212. get enumeration
// .971212. \\ ==> /   : fondamentale supporto di Windows NT / 95...
// .971214. Separator
//
```

```
package zuf;

import vice.*;
import java.util.*;
import java.io.*;



public class Config {
  static final int QUOTE = '"';
  static final int NDEFAULTS = 70;

  static protected Hashtable tab = null;

  public static String get(String key) {
    String result = (String) tab.get(key);

    if (result != null)
      return result;

    throw new ViceExitError(
      "(zuf.Config): Errore interno fatale.<br>Non trovo la chiave --" +
      key + "--"
    );
  }


    // modifica tab in base al contenuto di name
  public static void setUp(String name) {
    InputStream inStreamTmp;

    try {
      inStreamTmp = new FileInputStream(name);
      setUp(inStreamTmp);
      inStreamTmp.close();
    } catch (FileNotFoundException e) {
      if (tab == null) {
        tab = setUpDefaults();
        System.out.println("defs");
```

```
    }
    return;
  } catch (IOException e) {
    throw new ViceExitError(
      "(zuf.Config): Errore nella lettura di " + name + ".<br>" +
      e.getMessage()
    );
  }
}


public static void setUp(InputStream is) {
  AssertTokenizer inStream;
  tab = setUpDefaults();

  inStream = setUpStream(is);

  try {
    String parStr, valStr;

    while (inStream.nextToken() != inStream.TT_EOF) {
      parStr = loadName(inStream);
      valStr = loadValue(inStream);

      if (tab.containsKey(parStr)) {
        tab.put(parStr, valStr);
      } else
        throw new IOException(
          "Parametro inesistente alla linea " + inStream.lineno()
        );
    }
  } catch (IOException e) {
    throw new ViceExitError( "(zuf.Config): " + e.getMessage() );
  }
}



protected static Hashtable setUpDefaults() {
  Hashtable t = new Hashtable(NDEFAULTS);
  t.put("rootDir", "d:");
```

```
t.put("CatOpen", "newcat");
t.put("CatClose", "endcat");
t.put("SiteOpen", "sites");
t.put("SiteClose", "endsites");
t.put("Index", "index.swish");
t.put("Report", "report.txt");
t.put("ReportNew", "report.new");
t.put("SitesName", "sites.txt");
t.put("Extensions", ".html .htm .txt");
t.put("Catalog", "/vice/config/z.cat");
t.put("MailLog", "/zuff/quest/mail.txt");
t.put("MailHRef", "http://127.0.0.1/quest/mail.txt");
t.put("dbDir", "d:/db");
t.put("dbIndexDir", "d:/db");
t.put("cmpCmd", "d:/usr/local/bin/cmp.exe -s ");
t.put("cpCmd", "d:/usr/local/bin/cp.exe ");
t.put("idxCmd", "d:/vice/src/c/swish/swish.exe -c ./swish.cfg -i ");
t.put("searchCmd", "d:/vice/src/c/swish/swish.exe");
t.put("rmCmd", "d:/usr/local/bin/rm.exe -Rf ");
t.put("bakExt", ".bak");
t.put("SinOpen", "sinonym");
t.put("SinClose", "endsin");
t.put("HTMLSearch", "Tmpl/itree.html");
t.put("HTMLNavig", "Tmpl/newindx.html");
t.put("HTMLForm", "Tmpl/iform.html");
t.put("HTMLQuest", "Tmpl/quest.html");
t.put("HTMLSent", "Tmpl/sent.html");
t.put("HTMLAnsw", "Tmpl/answ.html");
t.put("HTMLResult", "Tmpl/result.html");
t.put("HTMLCSRUpdate", "Tmpl/csrupdate.html");
t.put("HTMLIndex", "Tmpl/index.html");
t.put("CSRIndex", "/zuff/csr/index.html");
t.put("Separator", "/");
t.put("FileOpen", "filebegin");
t.put("FileClose", "fileend");
t.put("CSRIdx", "e:/zk/csr");
t.put("CSRDir", "e:/home/vice/src/java/z/csr");
t.put("CSRUrl", "http://www.zuff.it/csr/");
t.put("questDir", "d:/quest");
t.put("questBakDir", "d:/questBak");
t.put("searchBakDir", "d:/zuff/searchBak");
```

```
    t.put("Semaphore", "_semaphore");
    t.put("FolderImg", "/zuff/image/closedfolder.gif");
    t.put("DocImg", "/zuff/image/doc.gif");
    t.put("StopWord", "stopwd.txt");
    return t;
  }


  protected static String loadName(AssertTokenizer in)
  throws IOException {
    if (in.ttype != in.TT_WORD)
      throw new IOException();

    return in.sval;
  }

  protected static String loadValue(AssertTokenizer in)
  throws IOException {
    in.nextToken();

      // il valore puo' essere solo un quote o una word
    if ((in.ttype != in.TT_WORD) && (in.ttype != QUOTE))
      throw new IOException();

    return in.sval;
  }


  protected static AssertTokenizer setUpStream(InputStream is) {
    AssertTokenizer result = new AssertTokenizer(is);
    result.eolIsSignificant(false);
    result.whitespaceChars(' ', ' ');  // portab. ?
    result.wordChars('$', '~');
    result.commentChar('#');
    result.quoteChar(Config.QUOTE);

    return result;
  }


    // for dumping purpose only...
```

```
  public static Enumeration elements() {
    return tab.elements();
  }

  public static Enumeration keys() {
    return tab.keys();
  }
}



//
// Find.java
//
// Andrea Vicentini 09/11/1997
//
// .971207. passando lista delle dirs
// .971213. prendo result di swish e ne faccio una lista
// .980105. salvo Sites da Category in OCategory per stampare label in out
//


package zuf;

import vice.*;
import java.io.*;
import java.util.*;



public class Find extends Suck implements Generator {

  protected String dirList;
  protected String kwdList;
  protected Catalog catalog;
  protected OTree otree;
  protected StopWordTable sw;

  protected static final String RESULTSTR = "[RESULT]";

  public Find(
```

```
    Catalog catalog, StopWordTable sw, String dirList, String kwdList
  ) {
    this.catalog = catalog;
    this.dirList = dirList;
    this.kwdList = kwdList;
    this.sw = sw;

    saveSearch(dirList, kwdList);

    String execCmd =
      Config.get("searchCmd") + " " +
      extractDir(dirList) + " " +
      extractKwd(kwdList);

    otree = new OTree("Risultati");

    try {
      if (! kwdList.equals(""))
        new BatchExec(execCmd, this, new SwishErrHandler());

      dump();
    } catch (IOException e) {
      System.out.println("IOERR: " + e.getMessage());
    }
  }

  protected String extractDir(String s) {
    StringBuffer result = new StringBuffer();
    StringTokenizer st = new StringTokenizer(s);
    String idx = Config.get("Index");

    while (st.hasMoreTokens())
      result.append(" -f " + st.nextToken() + "/" + idx);

    return result.toString();
  }


  protected String extractKwd(String s) {
    StringBuffer result = new StringBuffer(" -w");
    StringTokenizer st = new StringTokenizer(s);
```

```
    String tmp;

    while (st.hasMoreTokens()) {
      tmp = st.nextToken();
      if (! sw.contains(tmp))
        result.append(" (" + vString.stemmOn(tmp) + "*) and");
    }

    String r = result.toString();
    return r.substring(0, r.length() - 3);
  }


    // salva su file i parametri della ricerca
  protected void saveSearch(String catP, String kwdP) {
    try {
      FileOutputStream o = new FileOutputStream(
        Config.get("searchBakDir") + Config.get("Separator") +
        vString.replace( (new vDate()).toString(), ": ", '_' )
      );

      o.write( (catP + "\n" + kwdP).getBytes() );

      o.close();
    } catch (IOException ignored) { }
  }


  public void run() {
    ViceInputStream inStream;
    StringTokenizer dirStream = new StringTokenizer(dirList);
    String s;
    OCategory cur;

    try {
      inStream = new ViceInputStream(this.inStream);

        // skips 2 linee di commento con versione di swish
      inStream.readLine();
      inStream.readLine();
```

```
    while (dirStream.hasMoreTokens()) {
        // skips tutte le linee che iniziano con #
      while ((s = inStream.readLine()).charAt(0) == '#')
        ;

      if (!s.startsWith("err:")) {
        otree.mkBranch(
  catalog.labelsOf(dirStream.nextToken(), "|"), "|"
);
        cur = (OCategory) ((Node) otree.Current()).getObj();

        cur.addSites(
          ((Category)((Node)catalog.Current()).getObj()).getSites()
        );

          // poi stampo fino a trovarne ancora con '#'
        do {
          cur.addLine(s);
          s = inStream.readLine();
        } while ((s.charAt(0) != '#') && (s.charAt(0) != '.'));
      } else
        dirStream.nextToken();
    }

  } catch (EOFException e) {
    throw new ViceExitError("swish out format error.");

  } catch (IOException e) {}
}


  // stampa la lista di risultati
protected void dump() {
  Templater t = new Templater(Config.get("HTMLResult"));
  t.put(this);
  t.put(new SimpleGen("[KWD]", kwdList));
  try {
    t.go();
  } catch (IOException e) {
    throw new ViceExitError(e.getMessage());
  }
```

```
  }


  public boolean matches(String s) { return (s.indexOf(RESULTSTR) != -1); }
  public String explode() {
    return otree.htmllist();
  }
}

class SwishErrHandler extends Suck {
}



//
// GenTree.java
//
// Andrea Vicentini 07/12/1997
//


package zuf;

import vice.*;
import java.util.Enumeration;


public class GenTree implements Generator, Action {

  protected Catalog cat;
  protected boolean click;
  protected StringBuffer out;

  final public static boolean ClickOnFolder = true;


  public GenTree(Catalog cat, boolean click) {
    this.cat = cat;
    this.click = click;
    out = new StringBuffer();
  }
```

```
  public boolean matches(String s) {
    return s.equals("[TREE]");
  }


  public String explode() {
    String baseLabel =
      ( (Category) cat.getRoot().getObj() ).getLabel();
    String basePath =
      ( (Category) cat.getRoot().getObj() ).getPath();

    out.append("function generateTree() {\nUSETEXTLINK=1\n");
    out.append(
      "foldersTree = gFld(\"" + baseLabel +
      "\", \"" + basePath + "\");\naux0 = foldersTree;\n"
    );

    cat.getRoot().doAction(this);

    if (click)
      out.append(
"clickOnFolderRec(foldersTree, \"" + basePath + "\")\n"
      );

    out.append("}");
    return out.toString();
  }


  public void onTree(Object t) {
    onBoth((Category) t);
  }

  public void onLeaf(Object t) {
    onBoth((Category) t);
  }


  protected void onBoth(Category c) {
```

```
    if (c.getLevel() > 0) {
      out.append(
        space(c.getLevel()) +
        "aux" + c.getLevel() +
        " = insFld(aux" + (c.getLevel() - 1) +
        ", gFld(\"" + c.getLabel() + "\", \"" + c.getPath() + "\")) \n"
      );

      Enumeration e = c.getSnm();

      if (e.hasMoreElements()) {
        out.append("aux" + c.getLevel() + "[4] = new Array;");

        int n = 0;
        while (e.hasMoreElements()) {
          out.append(
            "aux" + c.getLevel() + "[4][" + n + "] = \"" +
            (String) (e.nextElement()) + "\"");

          n++;
        }
      }
    }
  }

  protected String space(int n) {
    StringBuffer result = new StringBuffer();
    for (; n > 0; n--)
      result.append(" ");

    return result.toString();
  }
}



//
// Mail.java
//
// Andrea Vicentini 26/12/1997
//
```

```
// .980105. passo per un file
// .980107. backup delle domande
//

package zuf;

import vice.*;
import java.util.*;
import java.io.*;

public class Mail {

  protected final static int NEXPERTS = 10;
  protected final static String DefaultName = "Senza titolo";

  Catalog c;

  public Mail(
    Catalog c, String dirList, String code,
    String title, String quest
  ) {
    Sync sema = null;
    String date = (new vDate()).toShortString();

    try {
      sema =
new Sync(Config.get("questDir"), Config.get("Semaphore"));

        // pulisco direttamente il titolo...
      title = vString.replace(title, " \"'`&<>", ' ');

      String name = getUnique(
        Config.get("questDir"), date + "_" +
        (title.equals("") ? DefaultName : title)
      );

      PrintStream outStream =
        new PrintStream(
          new FileOutputStream(
            Config.get("questDir") + Config.get("Separator") + name
          )
```

```
        );

      PrintStream outStreamBak =
        new PrintStream(
          new FileOutputStream(
            Config.get("questBakDir") + Config.get("Separator") + name
          )
        );

      Templater t = new Templater(Config.get("HTMLQuest"));

      t.put(new SimpleGen("[TITLE]", title));

      t.put(new GenCat(c, dirList));
      t.put(new SimpleGen("[QUEST]", quest));
      t.put(new SimpleGen("[CATEGORYHIDDEN]", dirList));
      t.put(new SimpleGen("[NAME]", name));
      t.put(new SimpleGen("[DATE]", date));
      t.put(new SimpleGen("[CODE]", code));

      t.go(outStream);
      t.go(outStreamBak);

      outStream.close();
      outStreamBak.close();

      (new Templater(Config.get("HTMLSent"))).go();

    } catch (IOException e) {
      throw new ViceExitError("(zuf.Mail):" + e.getMessage());

    } finally {
        // qualuque cosa succeda, e'
// fondamentale ripristinare il semaforo
      if (sema != null)
        sema.reset();
    }
  }


  protected String getUnique(String dir, String name) {
```

```
    File f;
    int n = 1;

    if (name == null)
      name = DefaultName;

      // tolgo i caratteri particolari
    name = vString.toAlNum(name, '_');
//    name = vString.replace(name, " :&`/\\*?|\"'<>", '_');

    String result = name + ".html";
    f = new File(dir, result);
    while (f.exists()) {
      n++;
      result = name + "_" + n + ".html";
      f = new File(dir, result);
    }
    return result;
  }
}



//
// OCategory.java
//
// Andrea Vicentini 04/01/1998
//
// .980105. doHrefWith tiene conto di dbDir
// .980106. immagini nella lista dei risultati
//


package zuf;

import vice.*;
import java.util.*;

public class OCategory implements Linkable {

  protected Node father;
```

```
  protected String label;
  protected Vector refs;
  protected Vector sites;

  public Node getFather() { return father; }
  public Object getKey() { return label; }
  public void setKey(Object s) { label = (String) s; }
  public boolean matches(Object s) { return label.equals( (String) s ); }

  public OCategory(Node n, String s) {
    father = n;
    label = s;
    refs = new Vector();
    sites = null;
  }


  public String getLabel() { return label; }


  public void addLine(String s) {
    refs.addElement(s);
  }


////////
//  Memorizza le informazioni (labels, ...) associate
//  alle categorie in <code>z.cat</code>.
//
  public void addSites( Enumeration e ) {
    sites = new Vector();
    while ( e.hasMoreElements() )
      sites.addElement( e.nextElement() );
  }


////////
//
//  Stampa la categoria formattata come una lista HTML.
//
```

```
  public String htmllist() {
    StringBuffer s = new StringBuffer(
      "<dt><image align=top src=\"" +
      Config.get("FolderImg") + "\">" + label
    );

    if (! refs.isEmpty()) {
      s.append("<dl>\n");

      Enumeration e = refs.elements();
      while (e.hasMoreElements())
        s.append(
          "<dt><image align=top src=\"" + Config.get("DocImg") +
  "\">" + doHrefWith((String) e.nextElement()) + "\n"
        );

      s.append("</dl>");
    }
    return s.toString();
  }



////////
//  (rank link title) diventa un link di nome title.
//  Data una linea di output di SWISH (rank link title) ne
//  costruisce una con un link, eventualmente modificando
//  la parte iniziale del link in <code>http://</code>.
//  Il title e' sostituito dalla
//  corrispondente label in <code>sites</code>
//  quando presente.
//
  protected String doHrefWith(String line) {

    if (line.startsWith("err:"))
      return "<it>" + line + "</it>";  // non dovrebbe mai capitare

    else {
      int linkIdx = line.indexOf(' ') + 1;
      int titleIdx = line.indexOf('"', linkIdx) + 1;
      int endTitleIdx = line.indexOf('"', titleIdx);
```

```
      // sistemo l'href, facendo la sostituzione se
      // per qualche ragione non l'ha fatta swish
    String href = line.substring(linkIdx, titleIdx - 2);
    if (href.startsWith(Config.get("dbIndexDir")))
      href = "http:/" + href.substring(Config.get("dbDir").length());

    boolean found = false;
    Site cur = null;

    if (sites != null) {
      Enumeration e = sites.elements();
      while (e.hasMoreElements() && ! found) {
        cur = (Site) e.nextElement();
        if (
          href.equals(cur.url) ||
          (
    cur.url.endsWith("/") &&
    href.equals(cur.url + "index.html")
  )
        ) {
          href = cur.url;
          found = true;
        }
      }
    }

    String title;
    String defTitle = line.substring(titleIdx, endTitleIdx);
    if ( found && (cur != null) && (cur.label != null) )
      title = cur.label;
    else if (! defTitle.equals("") )
      title = defTitle;
    else
      title = cur.url;

    return "<a href=\"" + href + "\">" + title + "</a>";
  }
}
```

```
////////
//  Calcola il titolo corretto per quel link.
//  Se trovo una label associata al link uso quella.
//  Altrimenti se <code>defTitle</code> non e' vuoto uso quello.
//  Altrimenti uso semplicemente il link.
//
  protected String getTitle(
    String link, Vector v, String defTitle
  ) {
    boolean found = false;
    Site cur = null;

    if (v != null) {
      Enumeration e = v.elements();
      while (e.hasMoreElements() && ! found) {
        cur = (Site) e.nextElement();
        if ( link.equals(cur.url) )
          found = true;
      }
    }

    if ( found && (cur != null) && (cur.label != null) )
      return cur.label;

    if (! defTitle.equals("") )
      return defTitle;
    else
      return link;
  }
}



//
// ONode.java
//
// Andrea Vicentini 04/01/1998
//
// .980105. String htmllist
//
```

```
package zuf;

import vice.*;


public class ONode extends MkNode {

  public ONode(TreeProducer p) {
    super(p);
  }

  public String htmllist() {
    StringBuffer s = new StringBuffer( ((OCategory) obj).htmllist() );

    if (getChild() != null)
      s.append("<dl>" + ((ONode) getChild()).htmllist() );

    if (getBrother() != null)
      s.append( ((ONode) getBrother()).htmllist() );
    else
      s.append("</dl>");

    return s.toString();
  }
}



//
// OTree.java
//
// Andrea Vicentini 04/01/1998
//
// .980105. htmllist string
//


package zuf;
```

```
import vice.*;
import java.util.*;


public class OTree implements TreeProducer {

  protected Object current;
  public Object Current() { return current; }

  protected ONode root;
  public ONode getRoot() { return root; }

  public Node newNode() { return new ONode(this); }
  public Linkable newObj(Node n) { return new OCategory(n, ""); }


  public OTree(String s) {
    root = new ONode(this);
    root.setObj( new OCategory(root, s) );
  }


  public Node mkBranch(String s, String sep) {
    current = getRoot().mkBranch(s, sep);
    return (Node) current;
  }


  public String htmllist() {
    if (root.getChild() != null)
      return "<dl>" + ((ONode) getRoot().getChild()).htmllist()
+ "</dl>";
    else
      return "Nessun risultato";
  }
}



//
// Site.java
```

```
//
// Andrea Vicentini 30/12/1997
//

package zuf;

public class Site {
  public int linkDepth = 0;
  public String label = null;
  public String url = null;

 public String toString() {
   return "( " + linkDepth + " " + label +  " " + url + " )";
 }
}
```