Consiglio Nazionale delle Ricerche

Architettura software, funzionalità e meccanismi del Sistema INFEA

Sabrina Tardelli, Loredana Versienti

Rapporto
CNUCE-B4-2002-004

CNUCE

Pisa

Architettura software, funzionalità e meccanismi del Sistema INFEA.

Sommario:

Il rapido e crescente sviluppo del World Wide Web come strumento di accesso ad Internet e la capillare trasformazione della tecnologia ad esso connessa ha generato una nuova rivoluzione informatica: utilizzando tecnologie a costi bassi molti nuovi soggetti hanno ora la possibilità di allargare le loro attività e distribuire informazione in un ambito più vasto. Un sistema informativo basato sul Web (WIS) rappresenta un ulteriore tassello della rivoluzione in atto; la potenzialità di raggiungere una comunità enormemente più ampia di quella di un sistema informativo tradizionale consente l'accesso ad informazioni e a funzioni anche ad utenti occasionali. Questo, assieme alla caratteristica di standard aperti e software spesso gratuito e facilmente disponibile, ha reso i WISs molto interessanti per l'ammodernamento delle istituzioni pubbliche offrendo un metodo non dispendioso di allargare le attività e di distribuire informazione ad un pubblico più ampio. Nel presente articolo vengono presentate le caratteristiche, le tecnologie e il funzionamento di un sistema informativo basato sul Web, denominato INFEA, realizzato per il Ministero dell'Ambiente.

1. Introduzione.

La diffusione crescente del World Wide Web come strumento di accesso ad Internet, determinato principalmente dalla comparsa di clienti multi-protocollo, sta generando notevoli cambiamenti nella gestione e organizzazione delle informazioni di moltissime attività. All'inizio il WWW è stato utilizzato, da aziende ed organizzazioni, principalmente come ambiente di diffusione delle informazioni. Nel corso degli anni i "siti Web" si sono raffinati ed arricchiti di servizi trasformandosi in strumenti sempre più complessi, che spesso affiancano il sistema informativo, là dove presente, o ne costituiscono il nucleo iniziale, sebbene a volte in maniera confusa. Nuove esigenze di applicazioni realizzate per il Web, ad esempio la possibilità di accedere basi di dati aziendali, e nuove possibilità derivanti da realtà emergenti, ad esempio l'e-Business, affiancati dalla maturazione di strumenti utilizzabili nel Web rendono sempre più necessario un ripensamento dei sistemi informativi per le organizzazioni.

Un sistema informativo basato sul Web (WIS – Web Information System) e' un sistema aperto in cui l'organizzazione dell'informazione ha una struttura più complessa rispetto a quella dei sistemi informativi tradizionali e la fruibilità dell'informazione è arricchita dalla natura intrinseca alla tecnologia Web.

Un'area molto importante per la realizzazione dei WIS è rappresentata dalle pubbliche istituzioni, enti che producono, raccolgono e distribuiscono una grande quantità di dati la cui conoscenza può risultare molto importante per cittadini e imprese. Tali sistemi sono chiamati Public Access WIS (PAWIS) ed hanno una caratteristica da cui non si può prescindere: l'accesso universale, cioè la possibilità di utilizzare i servizi o di accedere ai documenti pubblici deve essere fornita a tutti, a prescindere dall'esperienza e dalle dotazioni hardware dell'utilizzatore.

Nel presente lavoro, dopo una breve esposizione del rapporto tra Wis e istituzioni pubbliche, vengono discusse le caratteristiche fondamentali di un Wis particolare, denominato INFEA e realizzato per conto del Ministero dell'Ambiente: in particolare il paragrafo 3 darà una descrizione della tecnologia e dell'architettura software usata in INFEA mentre il paragrafo 4 illustrerà i meccanismi implementativi di funzionamento.

2. I WIS e le istituzioni pubbliche.

Le istituzioni pubbliche, amministrative e non, sono sicuramente quelle che possono ricevere i maggiori benefici dal crescente affermarsi dei sistemi informativi basati sul Web e della tecnologia che è alla base del loro sviluppo. Storicamente le pubbliche istituzioni sono state fino ad oggi le meno gratificate dalla crescente diffusione di strumenti informatici. Le ragioni possono essere svariate, a partire dalla complessità del modello organizzativo (ben altra cosa è, infatti, realizzare sistemi informatici per un'azienda di produzione, dove l'organizzazione stessa del lavoro è regolata dai cicli produttivi, rispetto alla realizzazione di un sistema per un ministero dove molti dei vincoli organizzativi sono regolati da burocrazia e leggi del parlamento), dalla carenza di un'idea guida di razionalizzazione, spesso penalizzata dal frequente cambio di dirigenze e d'indirizzi politici, o dalla scarsa penetrazione di cultura informatica, etc. In alcune situazioni, l'utilizzo di sistemi informatici ha generato un appesantimento piuttosto che uno snellimento dell'attività lavorativa, col risultato di scoraggiare piuttosto che ampliare l'interesse degli operatori verso la tecnologia. Molte cose negli ultimi anni sono cambiate, dall'istituzione di un'Authority per l'Informatica nella Pubblica Amministrazione, al crescente fabbisogno informativo, alla legge sulla trasparenza. Il rapido affermarsi di Internet ha dato la scossa più forte alla necessità di cambiamento nella gestione dell'informazione. Passata l'ubriacatura della presentazione di pagine Web per avere visibilità, le pubbliche istituzioni hanno cominciato ad avvertire la potenza che la rete delle reti forniva per la realizzazione di servizi per il cittadino e come potesse essere di grande aiuto alla semplificazione del lavoro all'interno dell'organizzazione. I WIS sono, secondo noi, una grande opportunità per l'organizzazione del lavoro nelle pubbliche istituzioni. Nelle aziende di produzione, il sistema informativo basato su strumenti di calcolo, è ormai una componente essenziale della stessa organizzazione, tale concetto è stato scarsamente utilizzato dalle organizzazioni pubbliche. La realizzazione di un WIS pone vincoli meno rigidi rispetti a quelli dei sistemi informativi tradizionali, basati su rete proprietaria, sia per le grandi capacità di scalabilità e di fruibilità intrinseche nella loro architettura che per la grande disponibilità di tools a basso costo, spesso free, indipendenti da hardware e software. Di là dalla progettazione, che deve seguire gli stessi rigorosi principi su cui si

basa la progettazione di un sistema informativo tradizionale, i WIS si prestano più facilmente alla realizzazione di componenti immediatamente fruibili; inoltre, nell'attesa di avere un sistema di componenti strettamente integrati, consentono l'utilizzo immediato di strumenti già disponibili nella tecnologia Web (ad esempio un sistema di posta elettronica), permettendo tempi di realizzazione più brevi e più facilmente graduabili nel tempo. L'accettazione stessa di un WIS, da parte degli utilizzatori, pone molto meno problemi rispetto a quella di un sistema informativo tradizionale, poiché le interfacce utenti, implementano metafore già largamente note agli utenti di Internet, oltre a fornire la possibilità di realizzare strumenti di ampia accessibilità. L'integrazione con legacy systems, là dove presenti, oltre alla semplificazione di realizzazione, dà l'opportunità di ripensare gradualmente all'organizzazione ed alla fruizione di funzionalità sempre più complesse per l'utenza interna all'istituzione, riducendo sensibilmente l'apprendimento nell'uso di strumenti informatici e ponendo le basi per la realizzazione di sistemi informativi per l'organizzazione.

3. Il sistema informativo INFEA.

Il sistema informativo INFEA è un esempio di Wis realizzato per la pubblica istituzione, il Ministero dell'Ambiente. L'obiettivo di INFEA è di fornire in maniera "integrata", sia strumenti di consultazione, che strumenti di gestione delle varie sorgenti di informazioni, con la possibilità di fruizione globale delle offerte informative dei vari enti coinvolti nel Sistema, come l'ISFOL, i centri regionali di EA, i laboratori territoriali e lo stesso Ministero. Inquadrare queste funzionalità in un nucleo di WIS è risultata una scelta naturale.

La Figura 1 mostra lo schema ad alto livello del sistema INFEA composto essenzialmente da un insieme di tipologie di servizi che accedono al database tramite richieste lanciate da utenti con ruoli e permessi differenti. Per una descrizione più accurata si rimanda a [Aloia00].

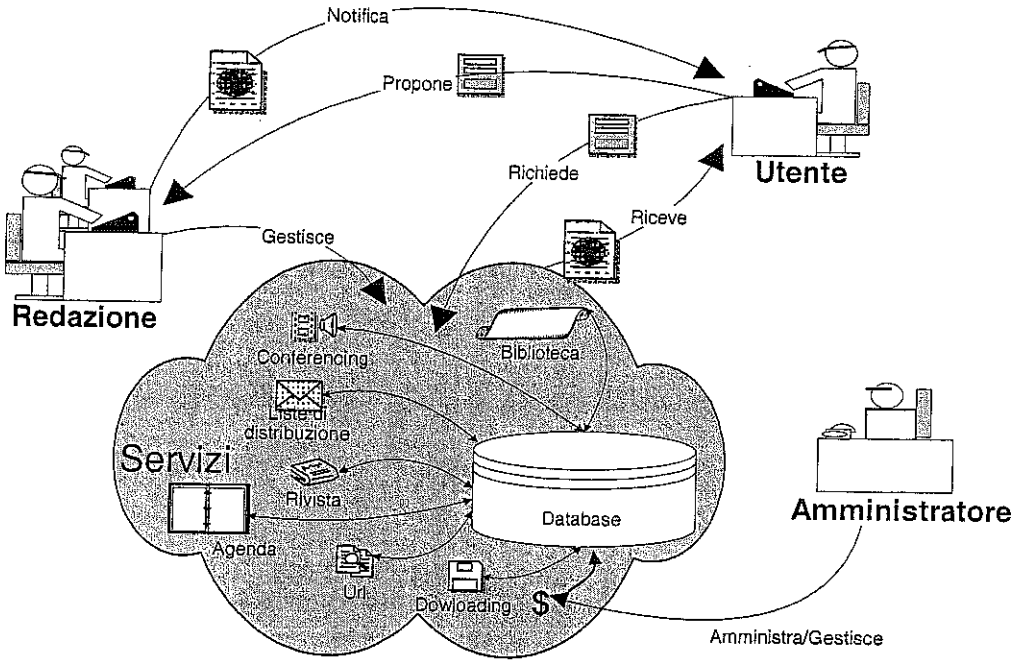

Figura 1 – Schema del sistema informativo INFEA.

3.1. Tecnologia usata in INFEA.

Nella realizzazione di WIS oggi è ampiamente utilizzata la tecnologia Java: si pensi agli applet o all'impiego di servlet e delle API JDBC [Aloia98] [Aloia99] per realizzare la *business logic* della componente *server-side* del sistema. Java si adatta bene alla realizzazione di WIS anche grazie ad un'altra caratteristica del linguaggio, la possibilità di accedere ad oggetti remoti (distributed object): è possibile cioè realizzare dei programmi in cui gli oggetti comunicano, tramite la rete, indipendentemente dalla loro localizzazione. L'utilizzo di tecnologie di gestione distribuita di oggetti, come CORBA (tramite il protocollo IIOP) e RMI (tramite i protocolli JRMP/IIOP), permette di realizzare delle applicazioni raffinate che consentono di superare in parte i limiti legati alla unidirezionalità del protocollo HTTP. Ad esempio è possibile implementare facilmente dei meccanismi di *call-back* tra server e client che evitino il polling continuo per l'accesso alle risorse. Questo impatta notevolmente sui tre sottosistemi che compongono un WIS:

Gestione della comunicazione: attraverso un uso combinato dei protocolli HTTP e IIOP o JRMP è possibile arricchire le caratteristiche di espandibilità e modularità dei WIS con la bidirezionalità e

l'accesso a metodi distribuiti.

Interazione uomo-macchina: utilizzando la tecnologia RMI o CORBA, gli applet Java diventano parte di un'applicazione distribuita, con i conseguenti vantaggi.

Gestione dei dati: utilizzando la comunicazione tra oggetti è possibile rendere più efficace e puntuale l'interazione con il gestore dei dati. Se il gestore dei dati, inoltre, utilizza la tecnologia ad oggetti, allora si ottiene il vantaggio del trattamento uniforme delle classi Java e delle classi del DBMS. [Aloia01]

Tutti gli oggetti del sistema INFEA sono realizzati mediante l'utilizzo del linguaggio Java, principalmente per garantire uno dei più importanti principi guida nella progettazione di INFEA, cioè l'indipendenza dalle caratteristiche hardware e software della piattaforma.

La tecnologia per la programmazione distribuita adottata in INFEA è RMI, ciò consente di superare molti limiti del protocollo http (e.g. trasmissione unidirezionale, mancanza di stati).

Il sistema ha un'architettura client/server a più livelli (multi-tier), in cui il primo livello consiste nell'interfaccia utente (*presentation logic*), il secondo livello comprende la logica dell'applicazione (*business-logic*) ed il terzo livello è costituito dal gestore dei dati (un DBMS). La *presentation logic* è implementata utilizzando il linguaggio Java, il formalismo HTML ed il linguaggio Javascript (che ha permesso di implementare anche parte della *business logic* sul lato client). La *business logic* e il *data access* sono implementati tramite applicazioni scritte in Java, mentre i dati sono memorizzati in un DBMS relazionale.

3.2. Architettura software di INFEA.

La figura 2 mostra l'architettura globale del sistema informativo INFEA, secondo il schema client/server multi-tier.

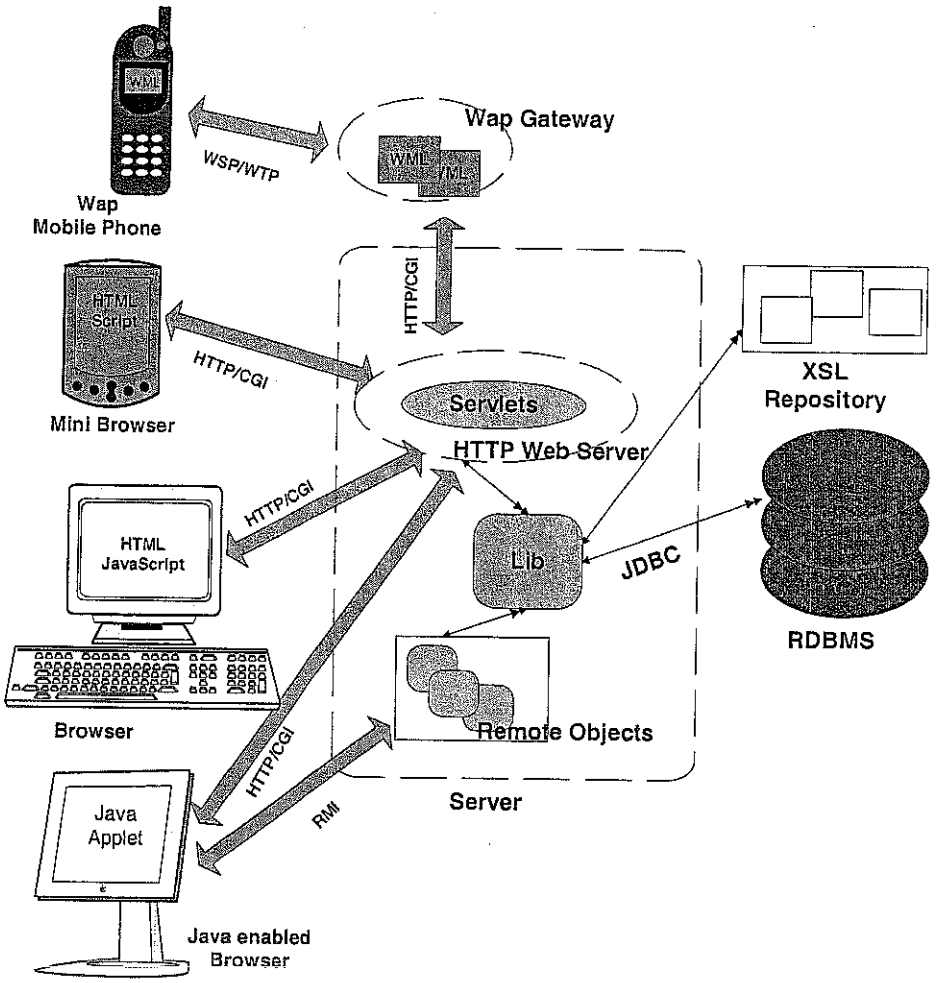

Figura 2: Architettura del sistema Infea

3.2.1. INFEA: Logica di presentazione.

Nel progettare la logica di presentazione del sistema INFEA, ci siamo posti l'obiettivo di definire interfacce semplici, intuitive e specifiche per il ruolo assunto dall'utente all'interno del sistema. Sono state definite tre tipologie di client:

- *Client basato su Applet Java.* È utilizzato nei casi in cui, oltre alla logica di presentazione, il client implementa anche una parte della business logic. Questa soluzione è stata adottata per le interfacce di inserimento, modifica e cancellazione dei dati, nelle quali oltre alla verifica della correttezza dei dati forniti, avvengono, in tempo reale, interazioni con il server per facilitare i compiti dell'utilizzatore. La comunicazione col lato server avviene tramite

l'invocazione di metodi remoti con il protocollo RMI di Java.

- *Client basato sul Web browser.* Questa tipologia è utilizzata quando la funzione preponderante del client è quella di presentazione. È stato utilizzato il formalismo HTML, sia per descrivere l'interfaccia per la formulazione della richiesta che per la restituzione dei risultati. Anche in questo caso sono implementate delle funzionalità minime di controllo sui dati per mezzo del linguaggio Javascript. La comunicazione col lato server avviene tramite il protocollo HTTP/CGI.
- *Client basati su dispositivi mobili.* Questa tipologia di client, fornisce alcuni servizi di interrogazione al sistema Infea tramite computer palmari e telefonini che supportano il protocollo WAP. In entrambi i casi l'implementazione si basa sui *microbrowsers* disponibili su questi dispositivi, utilizzando un sottoinsieme dell'HTML, nel caso di computer palmari, e del WML (Wireless Markup Language), nel caso dei telefoni mobili.

Nella figura 2, le immagini che compaiono sul lato sinistro rappresentano le tipologie di client; le doppie frecce che collegano questi componenti col riquadro centrale, rappresentante il lato server, mostrano i protocolli di comunicazione che essi utilizzano.

L'interfaccia (*client*) basata sul Web browser è rivolta alla comunità di utenti occasionali, spesso non esperti dei contenuti trattati dal sistema informativo, e/o nell'uso delle tecnologie informatiche. Per questa categoria di utilizzatori è stato curato molto l'aspetto interattivo, tramite interfacce più semplici ed intuitive, riducendo la complessità - e la potenza - dei meccanismi di ricerca delle informazioni. Un altro aspetto importante nel disegno di questa interfaccia ha riguardato la fruibilità. La realizzazione prevalentemente HTML, con l'ausilio di qualche piccola procedura Javascript, consente la massima fruibilità, in quanto non richiede nessuna operazione preliminare (come ad esempio l'installazione del plugin-Java per il cliente basato su applet) ed è utilizzabile su computer con capacità minima, sia dal punto di vista hardware che software. Il meccanismo di comunicazione col lato server è quello tradizionale fornito dal protocollo CGI: i dati rappresentanti la richiesta effettuata dall'utente vengono inviati ad un *servlet* Java che s'incarica dell'elaborazione e della generazione dinamica della pagina da restituire al client come risultato. In figura 3 è mostrato un

esempio di interfaccia realizzata con HTML e Javascript.

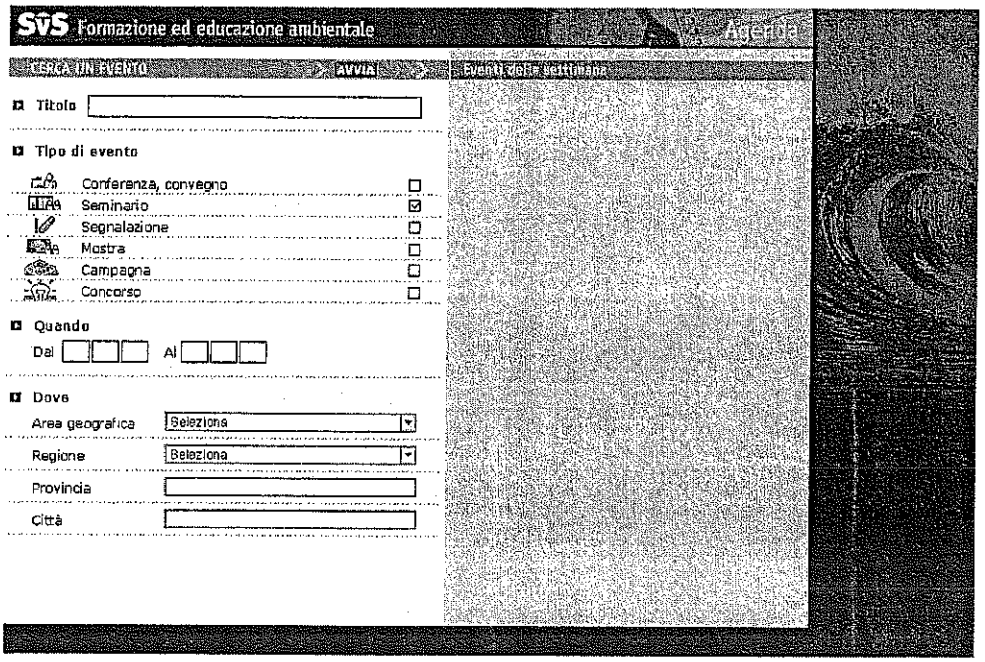

Figura 3: Esempio di interfaccia basata sul Browser.

L'interfaccia basata su applet Java è rivolta allo staff editoriale di INFEA e al management del servizio Sviluppo Sostenibile. Essa offre funzionalità di ricerca molto sofisticate, e funzioni per la gestione dei dati (inserimento, modifica e cancellazione). L'accesso tramite questa interfaccia è consentito solo attraverso un protocollo di autenticazione. Le specifiche tecniche per l'implementazione si basano sull'utilizzo delle classi Swing di Java per la realizzazione di un desktop in cui si possono attivare e disattivare finestre (la metafora della scrivania adottata dai moderni sistemi operativi). Sono state disegnate due modalità di comunicazione col lato server dell'applicazione: la prima invoca direttamente metodi remoti tramite il protocollo RMI e viene utilizzata in tutte le situazioni di passaggio di valori tra client e server (ad esempio l'inizializzazione di una lista dell'interfaccia con valori presenti nella base di dati); la seconda invoca un servlet tramite il protocollo CGI ed è utilizzata in particolari situazioni di visualizzazioni di informazioni (come ad esempio l'attivazione di un browser in una finestra Swing per mostrare informazioni dettagliate, Figura 4). Per l'utilizzo di questa interfaccia è necessario installare il Plugin-Java se non è già presente sulla stazione utilizzata.

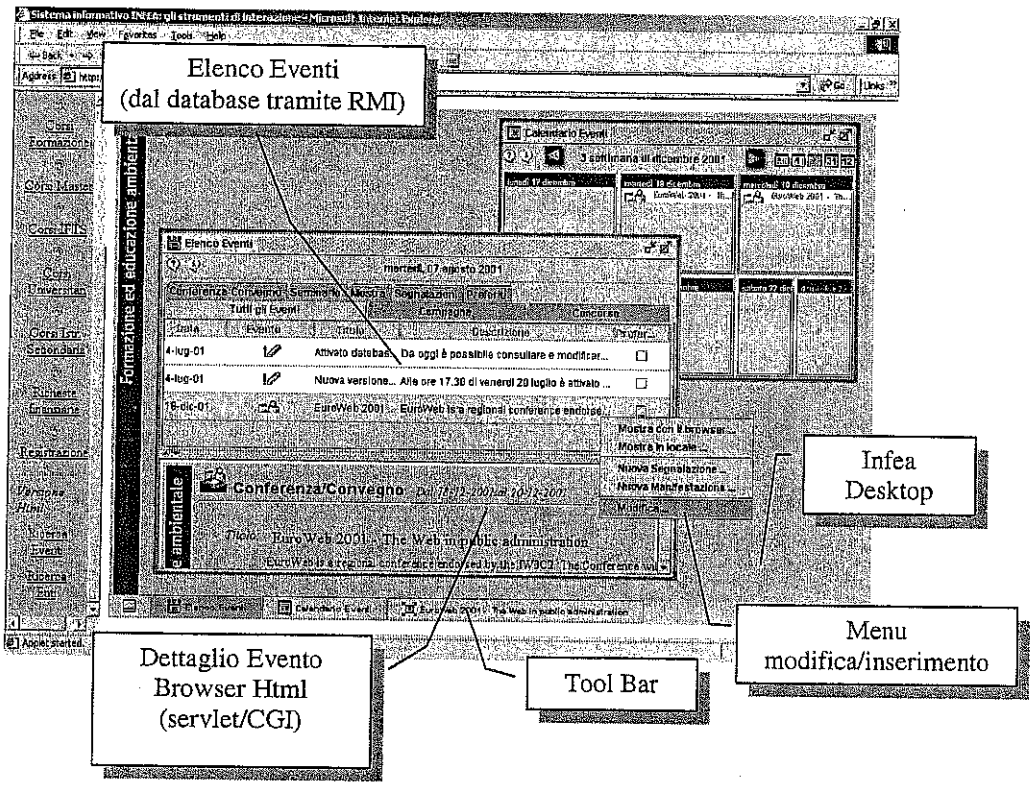

Figura 4: Esempio di interfaccia basata su applet Java.

L'interfaccia per dispositivi mobili è stata prevista nel sistema informativo Infea per eventuali sviluppi futuri di servizi avanzati. La realizzazione di un prototipo di client per computer palmari (PDA) con Windows CE e con telefonini Wap è stata svolta al CNUCE all'interno di un lavoro di tesi di laurea. La realizzazione è basata sull'utilizzo del sottoinsieme di HTML e di JavaScript supportato dal browser Explorer per Windows CE, nel caso del client PDA, e sul linguaggio WML nel caso dei telefoni mobili. Tramite il protocollo CGI viene invocato un servlet java che genera pagine di risultato che rispecchiano le caratteristiche fisiche del dispositivo mobile da cui arriva la richiesta. Le interfacce progettate sono più limitate per quanto riguarda la potenza di ricerca e sono state disegnate per essere funzionali col dispositivo in questione. In Figura 5 sono mostrate alcune componenti dell'interfaccia per la ricerca di enti realizzata sul palmare Compaq iPAQ. In Figura 6 sono mostrate alcune componenti dell'interfaccia per la ricerca di enti realizzata su telefoni mobili. In quest'ultimo caso sono stati realizzati due servizi: uno per la ricerca di enti con condizioni di ricerca ridotte (infatti, si possono esprimere condizioni solo su tipologia, nome e provincia dell'ente) ed uno per la ricerca di referenti (una forma di rubrica telefonica) su cui si possono esprimere condizioni sul nome della persona da ricercare sulla provincia, tipologia, e denominazione dell'ente di appartenenza.

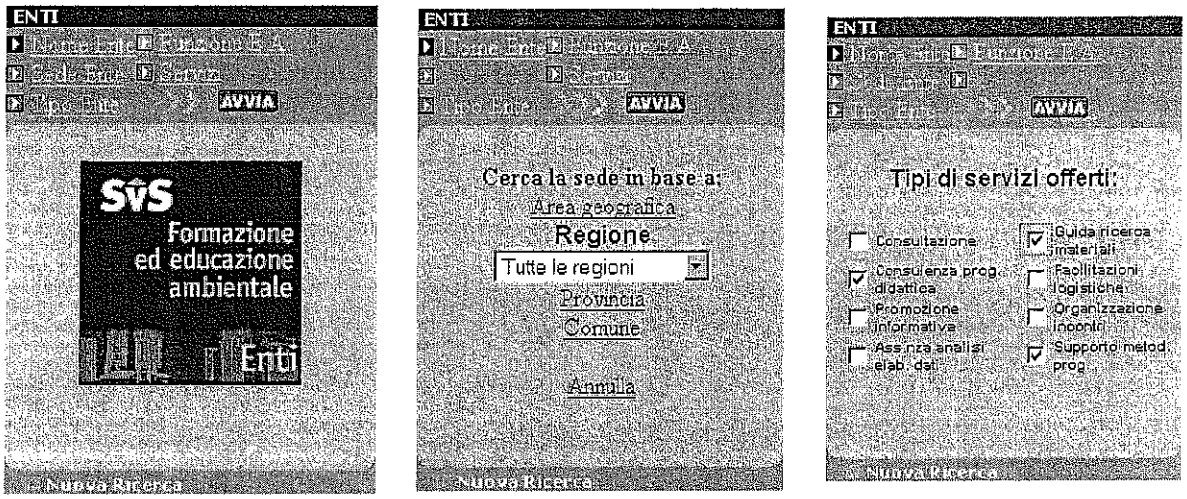

Figura 5 Esempio di interfaccia per computer palmari

Figura 6 Esempio di interfaccia per telefoni mobili

3.2.2. INFEA: Logica dell'applicazione.

Tutta la logica dell'applicazione è stata realizzata mediante l'utilizzo del linguaggio Java, principalmente per garantire uno dei più importanti principi guida nella progettazione di Infea e cioè l'indipendenza dalle caratteristiche hardware e software della piattaforma.

L'interazione con le varie tipologie di client descritte nel paragrafo precedente avviene mediante due componenti: *servlet* ed *oggetti remoti*. I servlet vengono invocati tramite il Web Server da un

qualsiasi client, mentre gli oggetti remoti sono invocati, tramite il protocollo RMI, solo da Client basati su Applet Java. L'invocazione di servlet da parte dei telefoni mobili arriva al Web Server attraverso il *Wap Gateway* con cui si è connessi (Figura 2). Sia i servlet che gli oggetti remoti condividono una libreria di classi per l'accesso al DBMS e per la trasformazione da XML a SQL e viceversa. In Figura 7 è illustrato con maggiori dettagli questo processo.

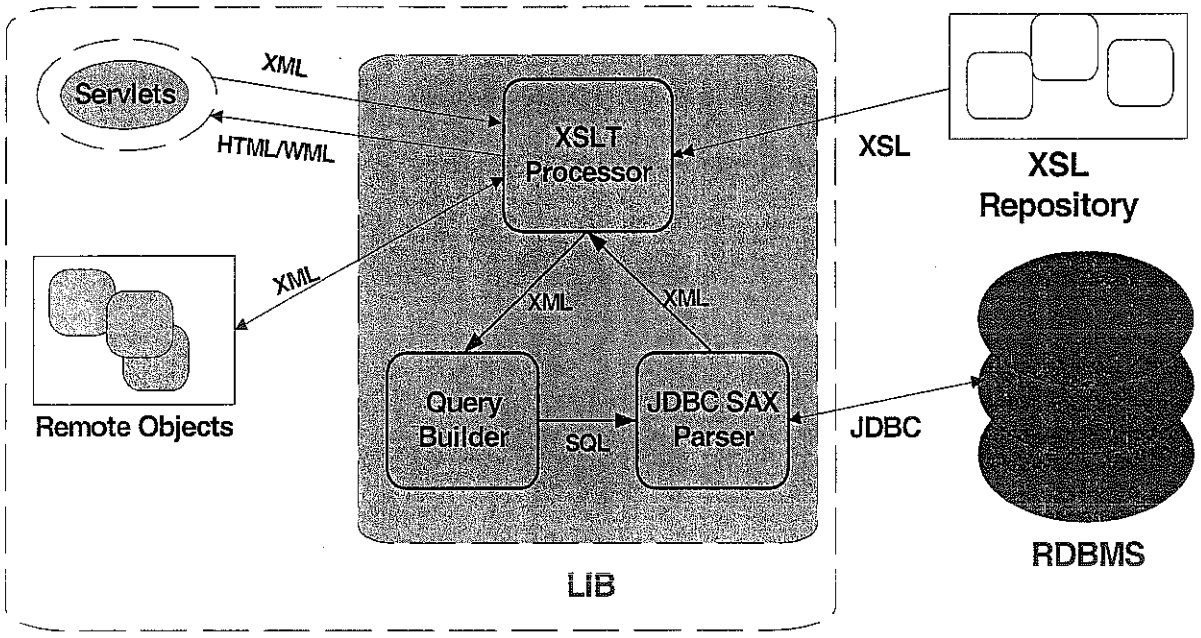

Figura 7 Processo di trasformazione

Sia dai servlet che dagli oggetti remoti, le richieste di informazioni, contenenti le varie condizioni di selezione, sono codificate in una stringa, utilizzando il linguaggio XML, che è inviata al modulo XSLT Processor. Questo modulo, tramite le informazioni contenute in un foglio di stile collegato alla richiesta, genera una stringa XML che viene inviata al modulo Query Builder. Il Query Builder trasforma la richiesta in una query SQL e la invia al modulo JDBC Sax Parser che invoca il DBMS, tramite le classe JDBC di Java, e trasforma il risultato dell'interrogazione in una stringa XML che viene restituita al modulo XSLT Processor. Il modulo XSLT Processor, basandosi su un foglio di stile che descrive le modalità di restituzione, genera il risultato per l'opportuno client.

Ricordiamo che i servlet eseguono esclusivamente funzioni di interrogazioni del DBMS, mentre gli oggetti remoti implementano, oltre alle funzionalità di ricerca, anche quelle di gestione e modifica

dei dati e di autenticazione dei client.

3.2.3. INFEA: Il gestore dei dati.

La maggior parte dei dati relativi al sistema informativo INFEA è raccolta da documenti fortemente strutturati; soprattutto per questo motivo abbiamo deciso di utilizzare un DBMS, come gestore dei dati. Per quanto riguarda la scelta del DBMS, abbiamo considerato la possibilità di utilizzare DBMS con modello dei dati ad oggetti (OODBMS), che avrebbero eliminato molti dei problemi di mapping tra il modello del documento con cui interagisce l'utente finale (interfaccia) ed il modello dei dati del DBMS. Lo stato attuale della tecnologia degli OODBMS, ancora non abbastanza matura ed affidabile, ci ha fatto propendere per l'utilizzo di DBMS con modello dei dati relazionale (RDBMS), che adottino SQL come linguaggio per la manipolazione dei dati (tecnologia affidabile e largamente utilizzata). L'accesso alla basi di dati avviene tramite le classi JDBC di Java, per cui non esiste la dipendenza da specifici RDBMS, purché adottino lo standard SQL (supportato dalla maggior parte dei fornitori di tali sistemi). Per la trasformazione dal modello dell'utente al modello del DBMS abbiamo adottato un approccio model-driven. Ogni documento XML è rappresentato da una struttura ad albero i cui nodi fanno riferimento alle entità della base di dati, le cui proprietà sono descritte dagli attributi dei nodi dell'albero. Il mapping è stato implementato utilizzando le classi SAX e JDBC di Java. Il processo di trasformazione è già stato illustrato in figura 7.

3.2.4. Considerazioni sull'architettura.

L'architettura multi-tier adottata comporta notevoli benefici sia nella gestione del sistema che nella manutenzione ed eventuali estensioni del software realizzato. Per esempio, qualora ce ne fosse la necessità, è possibile distribuire il carico del sistema su più macchine sia per problemi di sicurezza che di prestazioni (il DBMS può girare su una macchina diversa da quella su cui gira il Web server). È possibile cambiare o aggiungere modalità di presentazione senza la necessità di intervenire sul software che realizza la businnes logic. È possibile cambiare il DBMS (grazie all'utilizzo delle

JDBC) senza alcun intervento su tutto il software realizzato. Grazie alle funzionalità dei DBMS, sono garantite consistenza, privatezza e sicurezza dei dati. L'utilizzo di XML garantisce la portabilità dei dati tra le differenti componenti del sistema informativo e l'eventuale utilizzo con altri strumenti. I moduli software sono più facilmente mantenibili, ed eventualmente estesi, in quanto ognuno realizza solo le funzionalità richieste dal livello dell'architettura in cui si colloca.

Il lato server è stato progettato per gestire applicazioni multi-client con differenti livelli accessibilità, da quella tramite sofisticate interfacce grafiche a quelle testuali per telefonini mobili, creando un ambiente collaborativo basato sul paradigma noto come *Web based computing model.* L'utilizzo di oggetti distribuiti, permette di superare molti dei limiti delle applicazioni web tradizionali basate su HTML/HTTP, e di incrementare notevolmente le prestazioni e di ridurre il carico di rete.

La definizione di differenti categorie di client, oltre a fornire livelli di autorizzazione diversi, consente di utilizzare l'interfaccia più rispondenti alle risorse hardware/software di cui un utente dispone. Per ridurre il tempo di attivazione del client basato su applet java, l'applicazione è costituita da packages distinti che vengono scaricati solo al momento del bisogno; le dimensioni dei pacchetti vengono ridotte effettuando una compressione dei dati prima della comunicazione. La nostra esperienza di utilizzo, ha dimostrato che configurando opportunamente il plug-in Java (cioè abilitando jar caching, jtc, etc.) nell'ambiente del client, l'interfaccia fornisce buone prestazioni.

3.3. Progettazione delle interfacce utenti.

In collaborazione con gli addetti del Ministero, per ora ci siamo limitati ad analizzare due sole categorie di utenza: quella occasionale e quella esperta, a cui vengono assegnati privilegi diversi (la prima può solo consultare il contenuto informativo, la seconda può anche gestirlo, modificando inserendo e cancellando dati). I privilegi di *utente esperto* vengono assegnati dal ministero, previo analisi dei dati forniti tramite il modulo di registrazione, sviluppato per l'occasione. Per queste due categorie di utenza sono stati definiti i requisiti delle interfacce e fornite le specifiche di implementazione, risultanti dalla progettazione. Altre importanti funzionalità, quali la progettazione

e la realizzazione degli strumenti per le funzioni di staff editoriale e di amministratore, potrebbero in futuro essere oggetto di estensione all'attuale disegno del sistema informativo, a tale scopo l'architettura software è stata preposta a gestire tale situazione.

4. Descrizione dei meccanismi di funzionamento.

In questa sezione è fornita la descrizione dei meccanismi implementativi che rendono possibile la realizzazione delle varie richieste dei servizi offerti da INFEA, riferendoci di volta in volta all'organizzazione dei vari moduli che compongono il sistema e a dettagli implementativi delle classi coinvolte.

4.1. Organizzazione dei moduli.

L'insieme dei moduli è presentato nella sua struttura gerarchica. Tale gerarchia riflette l'organizzazione in *directory* (cartelle) utilizzata nella piattaforma che ospita il sistema ed è al tempo stesso l'organizzazione dei moduli adottata (Java *package*). In seguito il termine package e cartella verrà utilizzato come sinonimo. E' da notare, però, che le relazioni funzionali che intercorrono tra i vari moduli sono a volte trasversali a tale gerarchia. Il sistema è inizialmente suddiviso in due principali raggruppamenti. L'insieme dei moduli e delle risorse necessarie all'interazione con il *database*, e l'insieme dei moduli e delle risorse necessarie al funzionamento del server Web. Le cartelle preposte a tale scopo sono: **Classes** e **InfeaWeb**.

La directory ***Classes*** rappresenta la radice (root) di tutti i componenti software, ed è riferita sia dal codice remoto eseguito sul lato client (CODEBASE) sia dal codice locale eseguito sul server. Al suo interno si trovano tra l'altro le cartelle xslschema e Infea, più un insieme di file con estensione .jar (archivi) che vengono scaricati dal client, nel caso di esecuzione di applet Java.

La cartella **xslschema** contiene tutti i fogli di stile (file con estensione .xsl) utilizzati dal processore XSLT per effettuare le trasformazioni sui dati ricevuti/inviati dai vari componenti del sistema.

La cartella **Infea** rappresenta la radice di tutti i moduli Java sviluppati per il sistema informativo.

Vista come package, essa non contiene nessuna classe (.class) specifica, ma soltanto altri package che raggruppano i moduli in tre aree funzionalmente distinte: **client, common, server.**

Il package **infea.client** contiene tutto il codice necessario alla realizzazione dell'interfaccia utente basata su applet Java. Il suo contenuto, come descritto in precedenza, è archiviato in forma di file .jar nella cartella Classes. Ogni file .jar comprende i differenti moduli costituenti le varie componenti funzionali dell'interfaccia utente (agenda, enti, materiali, offerte educative, offerte formative, etc.).

Il package **infea.server** raggruppa tutte le classi e gli oggetti di cui è composto il Server RMI. La business logic del lato server è realizzata principalmente nel package infea.server.database, mentre i restanti package servono per il bootstrap del sistema.

Il package **infea.common** contiene tutte le risorse condivise tra il lato client ed il lato server, ed è anch'esso composto da vari package. In particolare i package codifiche e thesaurus realizzano l'associazione tra valori codificati nella base di dati (per le proprietà con valori codificabili) e la stringa presentata nell'interfaccia utente.

La directory *InfeaWeb* raggruppa tutte le risorse per il sito Web associato al sistema Infea. La struttura gerarchica segue le specifiche Java per la realizzazione di applicazioni Web basate su *servlet*. Secondo tali specifiche il package InfeaWeb è la radice dell'applicazione (sito) e corrisponde nella presente realizzazione al nome di dominio "/infea2001". A livello di radice sono posti tutti i file *html* per l'accesso al sito, mentre i servlet sono posti in *InfeaWeb\WEB-INF\classes*.

4.2. Funzionamento

4.2.1. Bootstrap.

Il processo di bootstrap del sistema è iniziato dalla classe *infea* del package infea.server.manager. Tale classe istanzia la classe *RMIServer*[1] appartenente allo stesso package, e la classe *InfeaProviderImpl* di infea.server.remote; quest'ultima a sua volta istanzia la classe *DBServiceImpl*.

[1] I sorgenti di Infea e RMIserver appartengono al progetto ServerManager in Z/Makes/infea/server/manager.

Queste classi sono implementazioni delle interfacce Java (*InfeaProvider* e *DBService*[2]), espongono le funzionalità del lato server e sono registrate tramite la classe *RMIServer*, in base al file di configurazione *InfeaSettings* presente nel package infea.common.

La classe *InfeaProvider* rappresenta il punto di accesso attraverso cui avvengono la registrazione di un nuovo utente, l'autenticazione dell'utente e l'attivazione di una connessione via RMI o via servlet con il sistema. Con l'istanziazione di *DBServiceImpl*, *InfeaProvider* demanda le funzioni suddette alla classe *UserManager* (package infea.server.database.users) che a seconda dell'azione dell'utente

- inserisce nel database il nuovo userId e Password (metodo *addUsers(...)*),
- inizializza la struttura dati con gli utenti abilitati (metodo *loadUsers(...)*),
- verifica che l'utente abbia i diritti per eseguire l'operazione richiesta (metodo *isAllowed(...)*).
- apre una nuova connessione (metodo *openConnection(...)*) nel caso la verifica precedente abbia esito positivo e l'utente non risulti già connesso.

Ancora attraverso l'istanziazione di *DBServiceImpl*, *InfeaProvider* demanda alla classe *InfeaObjectManager*[3] (package infea.server.database.objects) il compito di attivare in background lo *housekeeping thread* che nel corso della connessione controlla il tempo di lock degli oggetti in uso (metodi *lockObject(...)* e *unlockObject(...)*), esegue l'unlock degli oggetti bloccati da più di un prefissato intervallo temporale e libera tutti gli oggetti al momento della chiusura della connessione (metodo *unlockAllObjectsOf(...)*).

In figura 8 viene illustrata la gerarchia e la struttura delle classi coinvolte nel processo di bootstrap mentre la figura 9 ne rappresenta il diagramma di stati.

---

[2] I sorgenti di DBService, DBService_Impl, InfeaProvider e InfeaProvider_Impl appartengono al progetto ServerRemote in Z/Makes/infea/server/remote.

[3] I sorgenti di UserManager, InfeaObjectManager e KeyManager appartengono al progetto ResourceManagement in Z/ Makes/infea/server/ DBResourceManagement

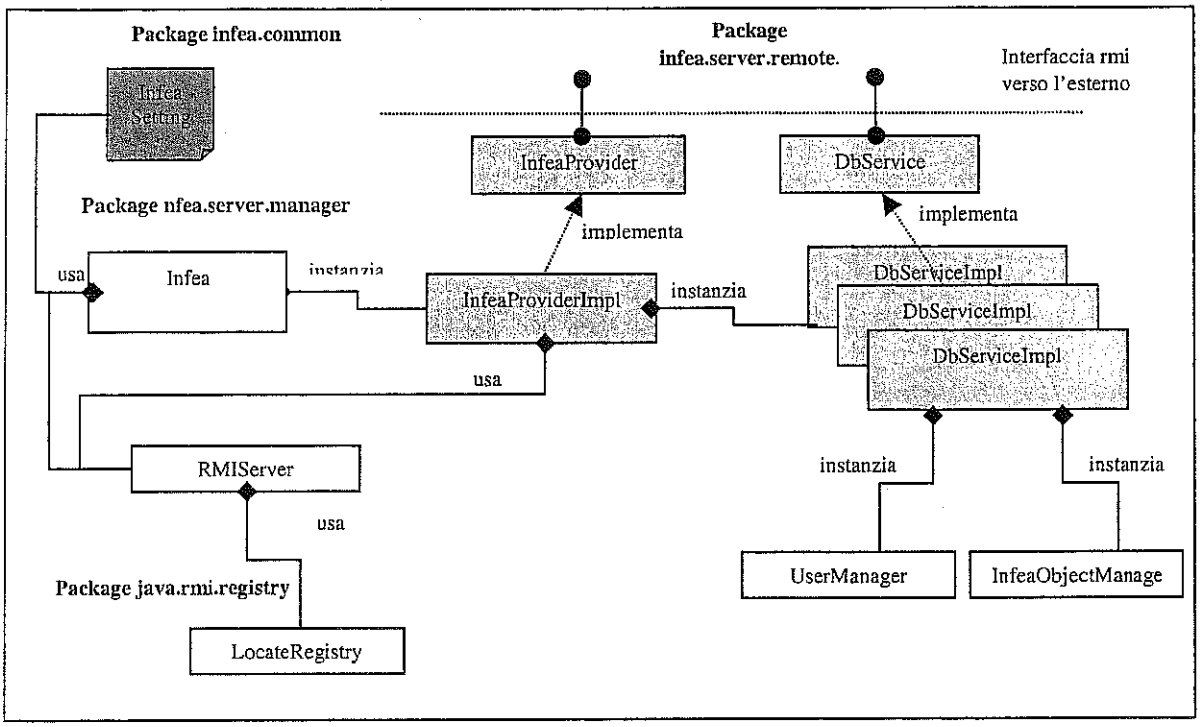

Figura 8 Gerarchia e struttura degli attori del meccanismo di bootstrap (modulo infea server).

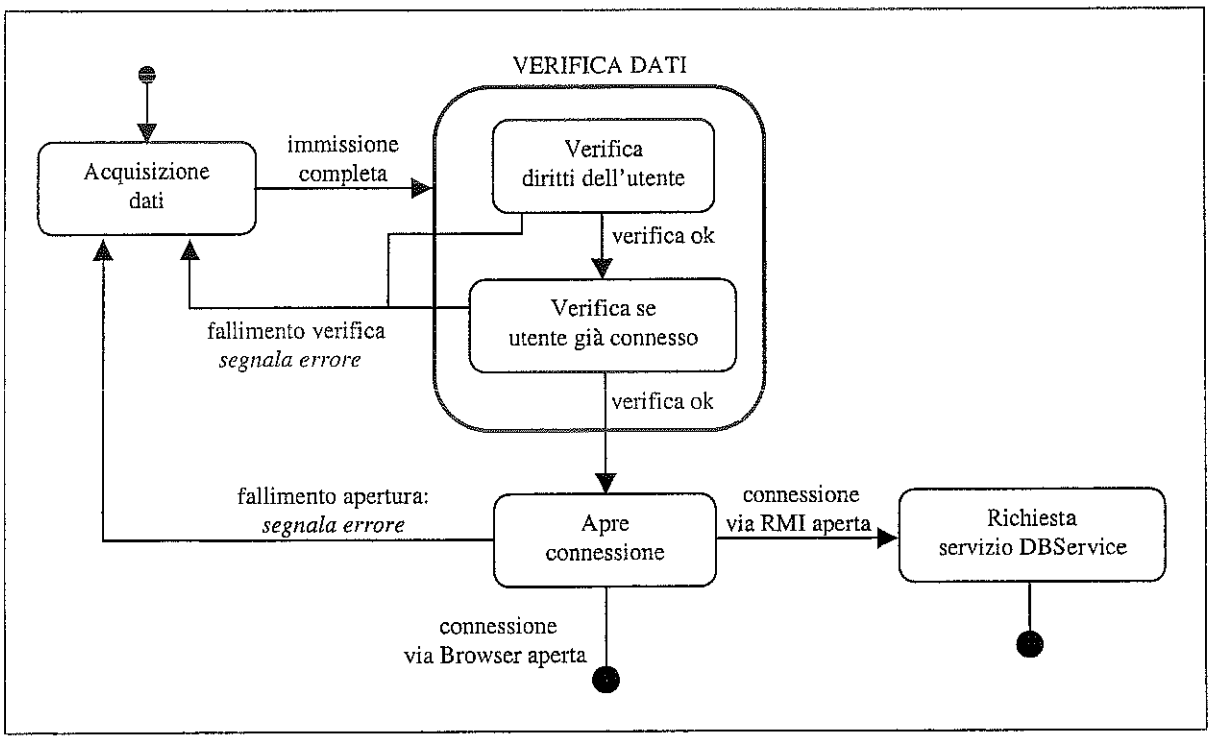

Figura 9 Diagramma degli stati del processo di bootstrap

4.2.2. Invio del NOME_UTENTE e PASSWORD: interazione con il server Web.

Per quanto riguarda l'interazione con il server web, una volta che l'utente invia userId e password, viene eseguito il servlet *VerificaServlet*. La classe *VerificaServlet* apre una sessione (richiamando *InfeaProvider* attraverso il metodo *openSession(...)* della classe *Common*) e verifica la validità dello userId e password. Se la verifica ha esito positivo contatta il *DBService* (attraverso il metodo *connectDbService(...)* della classe *Common*) e carica la pagina html contenente il desktop del sistema ("tools.html"), mentre nel caso in cui la verifica abbia esito negativo carica la pagina html contenente l'avviso di password errata ("Errore.html").

In figura 10 è rappresentato il diagramma degli stati del caricamento del desktop del sistema.

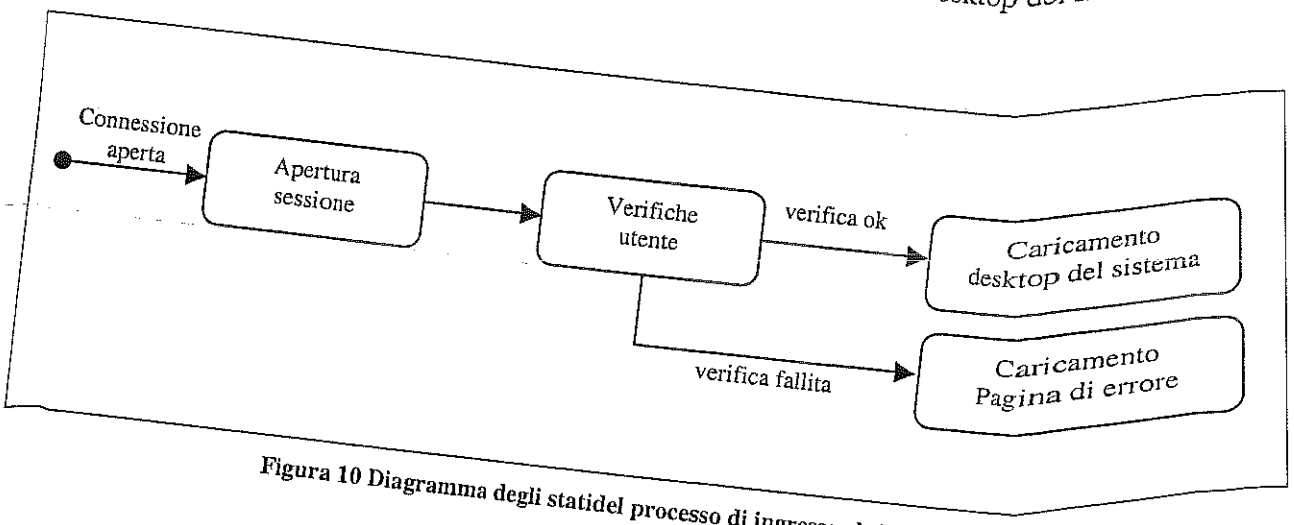

Figura 10 Diagramma degli statidel processo di ingresso al sistema

All'accesso al sistema contribuisce anche la classe *LoginServlet*[4] che individua il tipo di client da cui arriva la richiesta e si prepara a rispondere nel modo opportuno (a seconda del parametro *Target* ottenuto). Vediamo come agisce: se nella richiesta che arriva al server Web non è memorizzato un *cookie* relativo ad una sessione con Infea, *LoginServlet* richiede l'autenticazione del client a *VerificaServlet*, che controlla la validità di utente e password (metodo *isAllowed(...)*), servendosi di un metodo remoto del server RMI (metodo *isAllowed(...)* di *InfeaProvider*). Se l'autenticazione va a buon fine un *cookie* di sessione viene memorizzato sul client, viene contattato il *DBService* e viene

[4] I sorgenti di VerificaServlet, LoginServlet e Common appartengono al progetto InfeaLogin.vep in Z/ Makes/Web/Servlet/

inoltrata la pagina html relativa al modulo richiesto, che nel caso di client per utente esperto contiene l'invocazione all'applet opportuno. In figura 11 è illustrata la gerarchia e la struttura dei servlet citati.

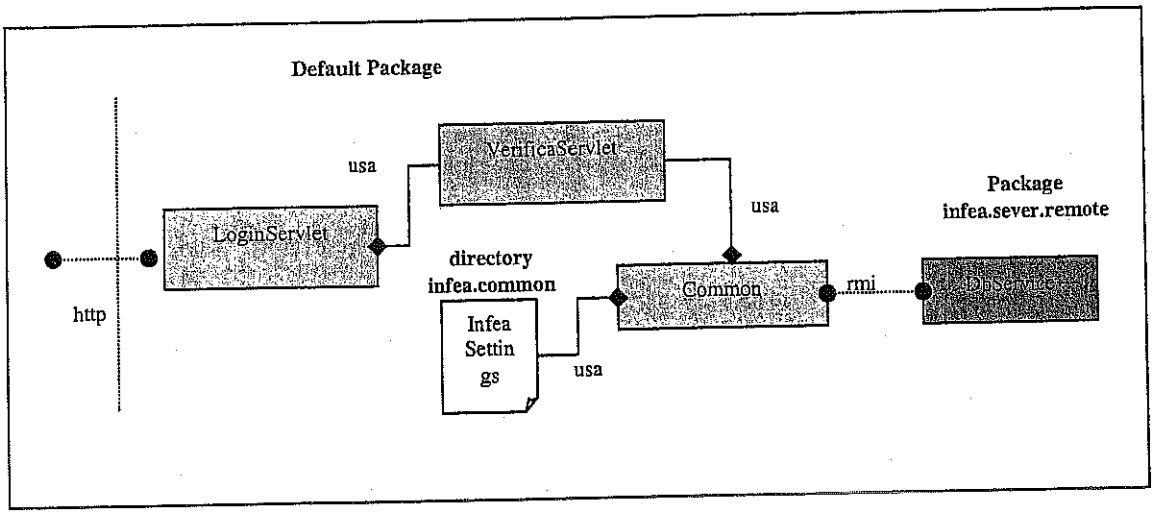

Figura 11 Gerarchia e struttura dei servlet Login e Verifica.

4.2.3. Ingresso nel sistema via applet: apertura di un modulo di ricerca.

Una volta stabilita una connessione e aperta una sessione valida, il sistema carica la pagina html ("tool.html") rappresentante il desktop di lavoro; a questo punto l'utente può usufruire delle funzionalità di INFEA accedendo a uno dei differenti moduli che costituiscono le varie componenti funzionali dell'interfaccia utente (agenda, enti, materiali, offerte educative, offerte formative, etc. – costituiscono il form "attivita.html").

Un click del mouse su una delle icone rappresentanti i moduli di ricerca provoca l'apertura dell'applet corrispondente, *Jappletmodulo*, assieme al caricamento sul client dei .jar relativi all'applet richiesto e del plug-in necessario al sistema che viene scaricato dal sito della Sun se il client ne è sprovvisto (tutto ciò, assieme all'invocazione dell'applet, avviene attraverso il foglio di stile *modulo_target.xsl*[5] lanciato da *buildAppletPage(...)* di *VerificaServlet*).

In figura 12 e 12a è rappresentato il diagramma degli stati dell'apertura di un modulo di ricerca.

[5] Il foglio di stile modulo_target è memorizzato in Z/InfeaWeb/WEB-INF/Classes.

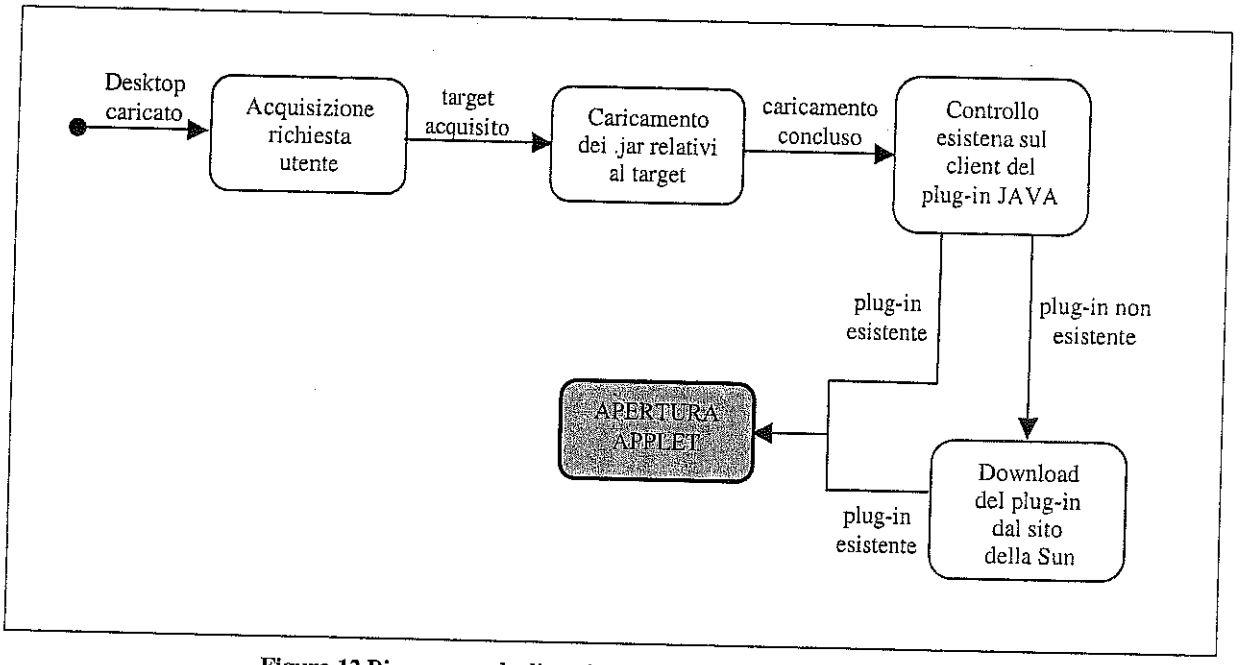

Figura 12 Diagramma degli stati di apertura di un modulo di ricerca

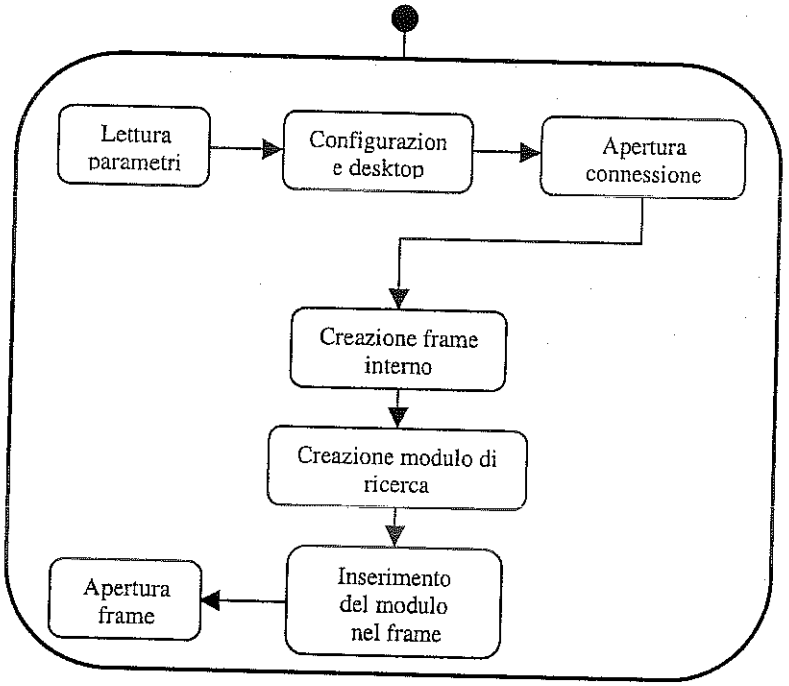

Figura 12a Diagramma dei sottostati di APERTURA APPLET

Tutti gli applet appartengono al package infea.client.applets e derivano dalla classe base *LBClientApplet*[6], package infea.client.common. Tale classe base è preposta a svolgere tutte le

[6] I sorgenti LBClientApplet, LBCommands, LBFrame e LBFilteredSearch appartengono al progetto Common in Z/Makes/infea/client/common.

operazioni che sono presenti nella fase di inizializzazione di ogni applet:

- lettura dei parametri – l'indirizzo dell'RMI, l'indirizzo dell'Host http, l'Odbc DataSource, l'Id della sessione, l'Id dell'utente, (metodo *doGetParameters()*);
- costruzione del Desktop (metodo *doConfigureDesktop()*);
- connessione RMI (metodo *doContactProvider()*), se la connessione viene aperta via RMI viene richiesta il servizio DBService (metodo *doRequestServices()*);
- caricamento delle codifiche e inizializzazione del Browser HTML (metodo *doPrepareExcute()*).

In più *LBClientApplet* lascia da implementare a *Japplet*Modulo il metodo *doExcuteCommands()* che si occupa dell'apertura della finestra di ricerca del particolare modulo. L'implementazione di tale metodo in *Japplet*Modulo consiste nella semplice chiamata di *open*Moduli*Search(...)* della classe *LBCommands* appartenente al package infea.client.gui.command.

*LBCommands.open*Moduli*Search(...)* istanzia la classe specifica del modulo funzionale di ricerca (*LB*Moduli*Search*, package infea.client.gui) come componente di un frame interno, *LBFrame* (estensione della classe java JInternalFrame), di cui esegue l'apertura e il settaggio dei parametri.

In Figura 13 viene rappresentata la gerarchia e la struttura del package infea.client.gui.

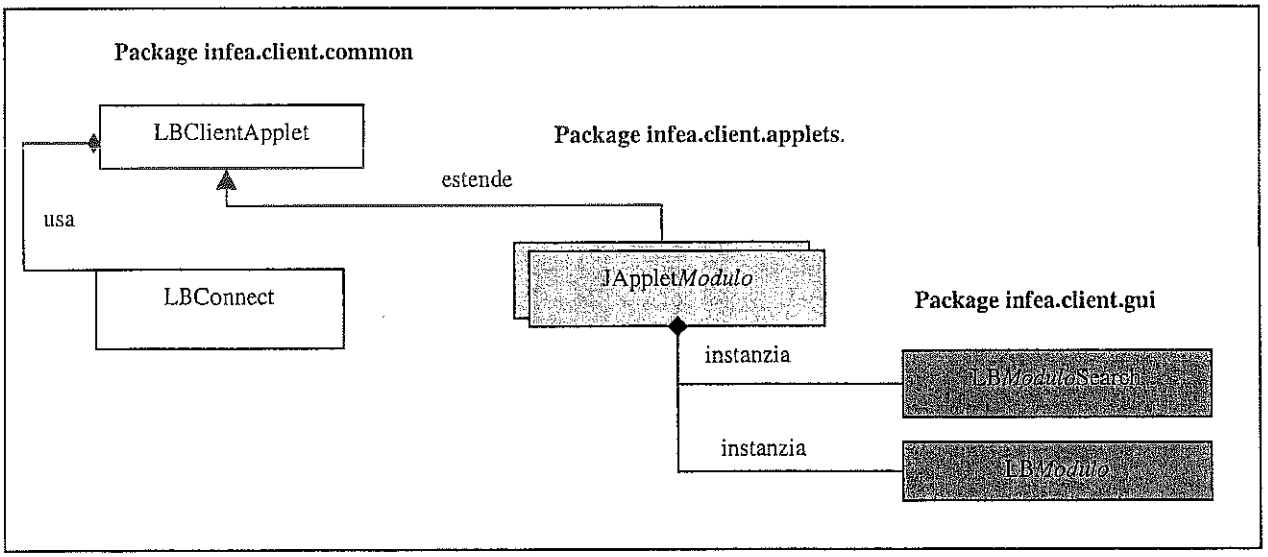

Figura 13 Gerarchia e struttura del package infea.client.gui

Le classi *LB*Moduli*Search* appartengono al package infea.client.gui e rappresentano una estensione

della classe astratta *LBFilteredSearch* (package infea.client.common) in cui viene definito il comportamento delle finestre di ricerca. Vediamole in dettaglio.

LBFilteredSearch estende JPanel e graficamente suddivide il pannello, tramite JSplitPane , in una sezione "filtri di ricerca" (metodi *mkFilters()* e *mkFilterButtons()*) e una sezione "lista dei risultati" (metodi *mkResultList()* e *mkResultListButton()*) e demanda alla classe concreta l'implementazione dei vari filtri e dei metodi che definiscono il comportamento del modulo.

*LB*Moduli*Search* estende *LBFilteredSearch* e tra l'altro svolge le seguenti azioni:

- pone a disposizione della classe base i vari filtri di ricerca (*LB*Modulo*Filter*) tramite il metodo *getFilters()*;
- definisce le intestazioni della tabella costituente la sezione "lista dei risultati" (metodo *getHeaders()*);
- raccoglie e struttura nel formato XML (metodo *getXmlQuery()*) i dati inseriti dall'utente nei vari filtri;
- raccoglie e visualizza graficamente nella sezione "lista dei risultati" (metodo *setXmlData()*) il risultato della ricerca

In figura 14 è illustrata la gerarchia e la struttura dei moduli di ricerca.

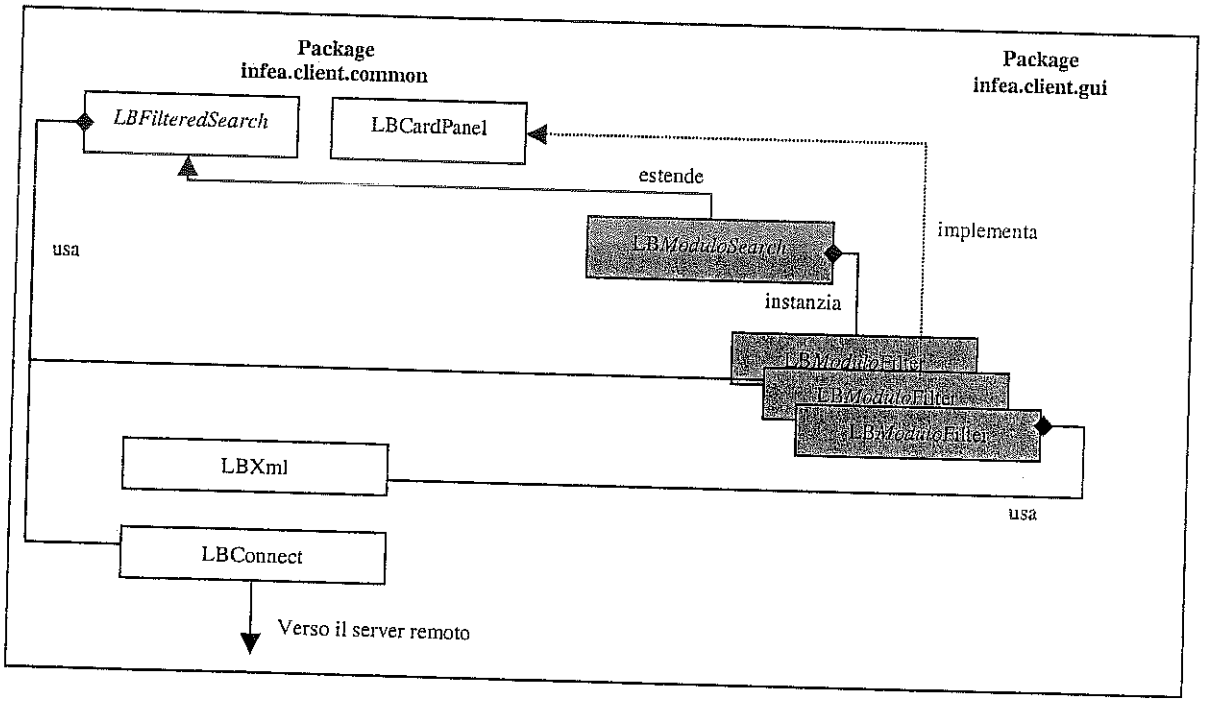

Figura 14 Gerarchia e struttura dei moduli di ricerca.

4.2.4. Invio di una ricerca

Una volta visualizzato l'intero modulo di ricerca l'utente è in grado di inserire i valori nei campi dei pannelli che costituiscono i filtri, e di richiedere una ricerca basata sulle condizioni poste. Il meccanismo di ricerca scatta appena l'utente preme il pulsante "*Cerca*": i dati inseriti dall'utente nei vari filtri vengono raccolti e strutturati nel formato XML (metodo *getXmlQuery()* di *LB*Modulo*Search*) incorporandovi altre informazioni di servizio quali lo userId, il tipo della richiesta *LIST_REQUEST*, il tag associato al tipo di richiesta *ELENCO*MODULI, l'Id della sezione, il nome del database. Tali dati, assieme, costituiscono la richiesta di ricerca (oggetto *RequestDescriptor*) e vengono passati, attraverso l'output di uscita (la Java Console), dalla classe *LB*Modulo*Search* alla classe *DBService*[7] (package infea.client.common) per mezzo del metodo *SendQuery(...)*.

DBService trasferisce l'oggetto ricevuto alla classe *DbService_Impl* affinché questa possa eseguire la query (metodo *eseguiQuery(...)*), e si pone in attesa della risposta.

*DbService_Impl.eseguiQuery(*request*)* esegue vari controlli prima di ottenere il risultato della richiesta:

- controlla che l'utente sia abilitato, attraverso il metodo *isAllowed* della classe *UserManager*. Ad esempio viene controllato che l'utente sia conosciuto, che non sia già connesso su un altro host, che abbia i diritti per richiedere un certo tipo di operazione (un utente generico non può richiedere modifiche sui dati). Tali controlli vengono eseguiti sfruttando le informazioni del vettore *RequestDescriptor:* l'userId e il tag associato al tipo di operazione richiesta – ricerca/modifica/inserimento/cancellazione;
- controlla che l'oggetto o gli oggetti coinvolti nell'operazione siano disponibili, cioè non siano usati in modalità di modifica da altri utenti; tale controllo è necessario per mantenere la consistenza dell'informazione e avviene attraverso il metodo *isObjectOwnedBy(...)* e *lockObject(...)* della classe *InfeaObjectManager*, package infea.server.database.object.

Se tali controlli risultano positivi, *DbService_Impl* applica una prima parziale trasformazione dei dati

[7] Il sorgente DBService del package common appartiene al progetto Common in Z/Makes/infea/client/common e non va confuso con l'omonimo sorgente del package infea.server.remote.

ricevuti dal client in formato XML (metodo *applyInputStyles(...)*):

- attraverso la classe *Input_StylesCatalog*[8], *DbService_Impl* recupera il path e il nome del foglio di stile, *SelezionaElenco*Modulo.*xsl*, associato al tag della richiesta (*ELENCOMODULO*).
- attraverso il metodo *applyXSL(...)*, *DbService_Impl* passa il path alla classe *XMLTransformer*, package infea.server.database, che si occupa di far processare il foglio di stile al processore XSLT.

DbService_Impl trasferisce il risultato di questa prima trasformazione ad una classe che in tale processo svolge un ruolo importante, *QueryBuilder*[9], package infea.server.database.updater.

QueryBuilder è una estensione della classe *HandlerBase* e trasforma le richieste, pervenute in formato XML, in un vettore di operazioni SQL: costruisce un SAXParser a cui chiede di eseguire il parse dell'input (dato in formato XML), seguendo i metodi di handler implementati dalla classe stessa: troviamo sovrascritti i metodi *startDocument()*, *endDocument()*, *startElement(...)*, *endElement(...)*, *characters(...)*. Tramite l'implementazione di tali metodi *QueryBuilder* in base agli elementi e agli attributi che incontra nel documento, riempie una serie di strutture dati ognuna predisposta a contenere elementi che avranno lo stesso ruolo nelle operazioni SQL, ad esempio il vettore che contiene i nomi delle tabelle coinvolte nelle operazioni, il vettore contenente gli statement, e così via. Al raggiungimento della fine del documento, *QueryBuilder* costruisce il vettore di query SQL attingendo opportunamente a tali strutture dati.

Vediamo alcune funzionalità dei metodi suddetti

StartElement(...) analizza ogni elemento della richiesta (la parola elemento è usata nel significato del formalismo XML). Un elemento nella nostra convenzione può

- iniziare con un segno di *underscore* il che significa che l'elemento rappresenta il nome di una tabella e come tale viene memorizzato in un particolare vettore (*vFrom*) i cui elementi nella query occuperanno il posto dopo la clausola FROM,

---

[8] I sorgenti Input_StylesCatalog, Output_StylesCatalog e XMLTrasformer appartengono al progetto Server-Xml in Z/Makes/infea/server/DBXml .

[9] I sorgenti QueryBuilder, JDBCSAXParser, JDBCInputSource, ParserBase, DataSelector e DataUpdater appartengono al progetto DataManipolation in Z/Makes/infea/server/DBDataManipolation

- essere uguale alla parola "*UPDATE*", "*SELECT*" e "*INSERT*", in questo caso rappresenterà lo statement della query,
- essere uguale alla parola "*ORDERBY*", in questo caso avremo un elemento con degli attributi che corrisponderanno al nome delle colonne in base alle quali bisognerà richiedere l'ordinamento,
- essere uguale all'operatore "*OR*" incluso in una richiesta di select, in questo caso bisognerà memorizzare in un vettore particolare una parentesi. Gli elementi che via via verranno inseriti in tale vettore sono quelli che costituiranno l'argomento dell'operatore *OR* nel corpo della clausola WHERE.

E ancora, se l'elemento contiene attributi, tali attributi possono, sempre secondo la nostra convenzione, essere le chiavi delle entità da aggiornare o da inserire, *QueryBuilder* le riconosce come tali e le memorizza in una struttura opportuna.

Quando il parser arriva alla fine di un elemento scatta il metodo *endElement()*. Vediamone alcuni aspetti. Se l'elemento di cui si è raggiunta la fine è un "*OR*" nello stesso vettore in cui era stata aperta una parentesi ora viene chiusa; se l'elemento rappresentava una tabella e lo statement era un comando di "*UPDATE*", al vettore che verrà poi ripercorso per ricostruire la query viene aggiunto lo statement "*SET*", seguito dal nome della tabella e dallo statement "*WHERE*", seguito dalla serie di nomi memorizzati nel vettore costituente la clausola where (*sbkeys*). Se l'elemento era una "*INSERT*", viene eseguito lo stesso procedimento precedente dove al posto degli statement "*SET*" e "*WHERE*", viene inserito lo statement "*INTO*".

Quando *QueryBuilder* raggiunge la fine del documento si troverà una serie di strutture dati che dovrà semplicemente svuotare in un certo ordine per ricavare gli argomenti con cui formare le query (metodo *endDocument()*).

Una volta ricavato il vettore di query dal *QueryBuilder*, *DbService_Impl* utilizza la classe *DataSelector* per ottenere il risultato della query (metodo *doDataSelection(...)*). Tale classe appartiene al package infea.server.database.selector, contiene la logica del gestore dei dati e fa uso della classe *DBConnectionBroker* che gestisce un pool di connessioni col database. In particolare *DataSelector* gestisce tutte le richieste di selezione dei dati, basandosi sulla classe *JDBCSAXParser*,

(questa classe rappresenta il parser SAX per database, estende la classe astratta *ParserBase* che implementa l'org.xml.sax.Parse e che gestisce solo pochi *handlers*).

JDBCSAXParser utilizza la classe *JDBCInputSource* (parser che opera sul JDBC data source) per generare eventi SAX durante l'iterazione su ogni riga e colonna della tabella del database con lo scopo di convertire in formato XML il ResultSet ottenuto, vediamo come.

Tramite il metodo *getConnection(...)* di *JDBCInputSource*, *JDBCSAXParser* stabilisce una connessione con il database e dopo aver creato un comando SQL, tipicamente una SELECT (metodo *createStatement()* dell'interfaccia *Connection*, package java.sql), lo esegue ottenendo un ResultSet (metodo *executeQuery(...)* dell'interfaccia *Statement*, package java.sql).

JDBCSAXParser trasforma quindi il ResultSet in un flusso di dati in formato XML: scorre il ResultSet ciclando, una prima volta, sui nomi delle tabelle, e una seconda volta sulle colonne di ogni tabella, scrivendo per ogni colonna con valore diverso da *null*, il nome che la identifica seguito dall'attributo *val*. Il risultato è illustrato nella seguente tabella:

```
<_NOMETABELLA1 >
    <PrimaColonna val=valore/>
    <SecondaColonna val=valore/>
...
</_NOMETABELLA1 >

<_NOMETABELLA2 >
    <PrimaColonna val=valore/>
    < SecondaColonna val=valore/>
...
</_NOMETABELLA2 >
        ....
```

Finalmente il flusso di dati, consistente nel risultato della query trasformato in XML, arriva attraverso la classe *DataSelector*, a *DBService_Impl*, che per mezzo del metodo *applyOutputStyles* applica un'ulteriore trasformazione:

- utilizzando la classe *Output_StylesCatalog*, *DBService_Impl* recupera il path e il nome del foglio di stile associato al tag specifico che accompagna la richiesta (*ELENCO*MODULI), (tale foglio di stile ha nome *Elenco*Modulo.*xsl*).
- attraverso il metodo *applyXSL(...)*, *DBService_Impl* passa il path alla classe *XMLTransformer* che si occupa di far processare il foglio di stile al processore XSLT.

Il risultato di quest'ultima trasformazione viene passato alla classe *DBService* da cui era scaturito l'intero meccanismo di ricerca. *DBService* si occupa di trasferire tali informazioni sull'output di uscita (la Java Console) da cui vengono raccolte dalla classe *LB*Moduli*Search*. *LB*Moduli*Search* attraverso il metodo *setXmlData(...)* ripercorrendo la struttura dei dati ricevuti, estrae le informazioni richieste dall'utente visualizzandole graficamente in posizione opportuna nella tabella costituente la sezione "lista dei risultati".

A questo punto il sistema si ripone in attesa di un'ulteriore azione dell'utente.

In figura 15 viene presentato il diagramma degli stati del meccanismo di ricerca, i rettangoli azzurri verranno espansi nelle figure successive.

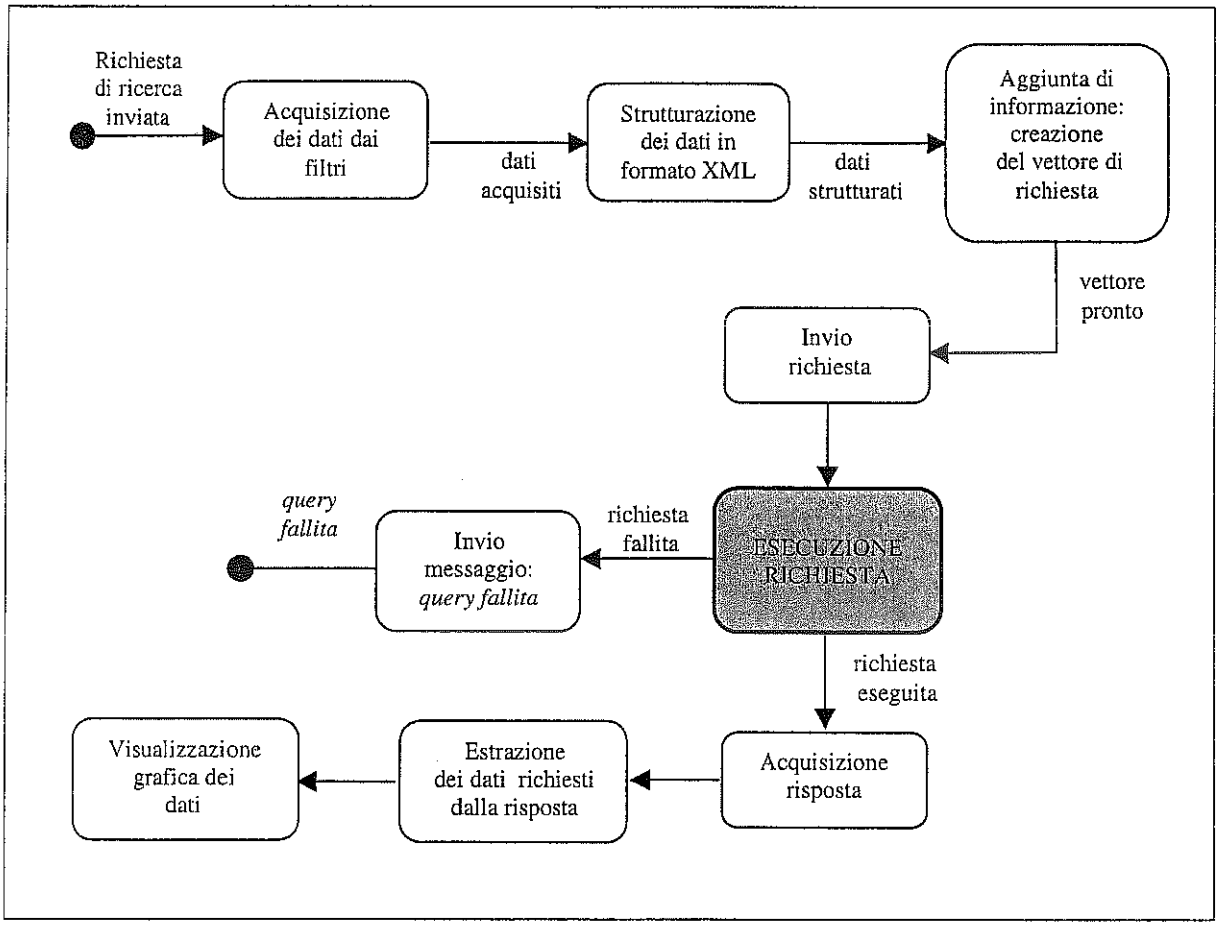

Figura 15 Diagramma degli stati del processo di ricerca

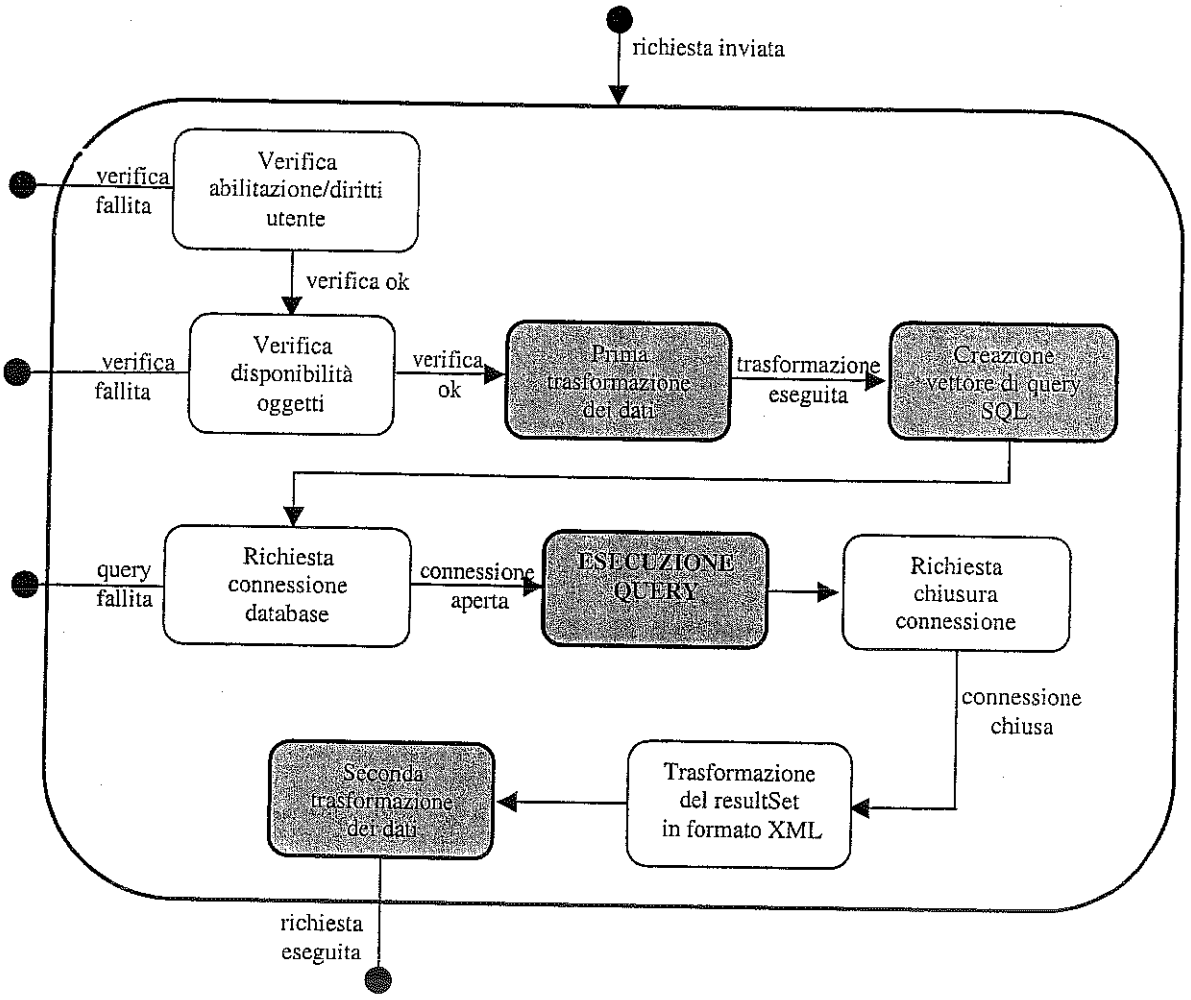

Figura 15a Diagramma dei sottostati di ESECUZIONE RICHIESTA

La prima e seconda trasformazione seguono lo stesso procedimento differendo soltanto per la classe utilizzata per l'acquisizione del path e del nome del foglio di stile.

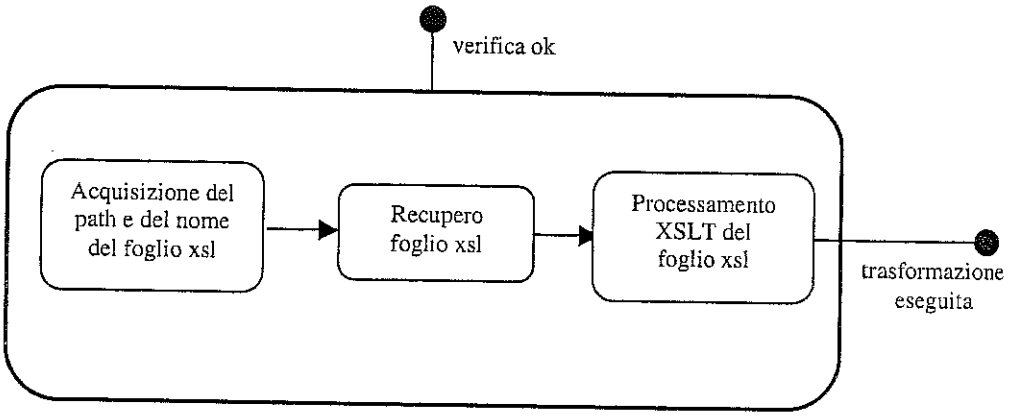

Figura 15b Diagramma dei sottostati di PRIMA/SECONDA TRASFORMAZIONE DEI DATI

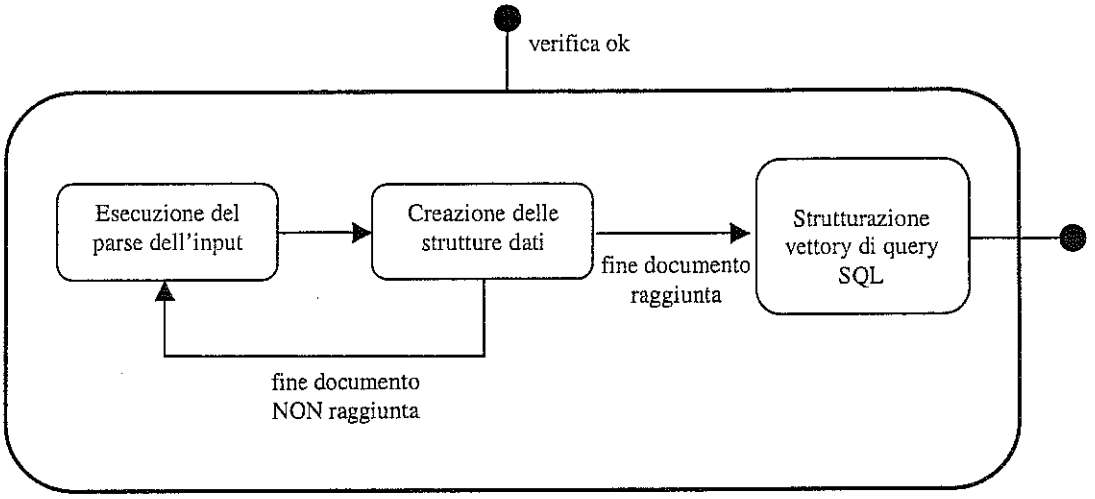

Figura 15c Diagramma dei sottostati di CREAZIONE DEL VETTORE DI QUERY SQL

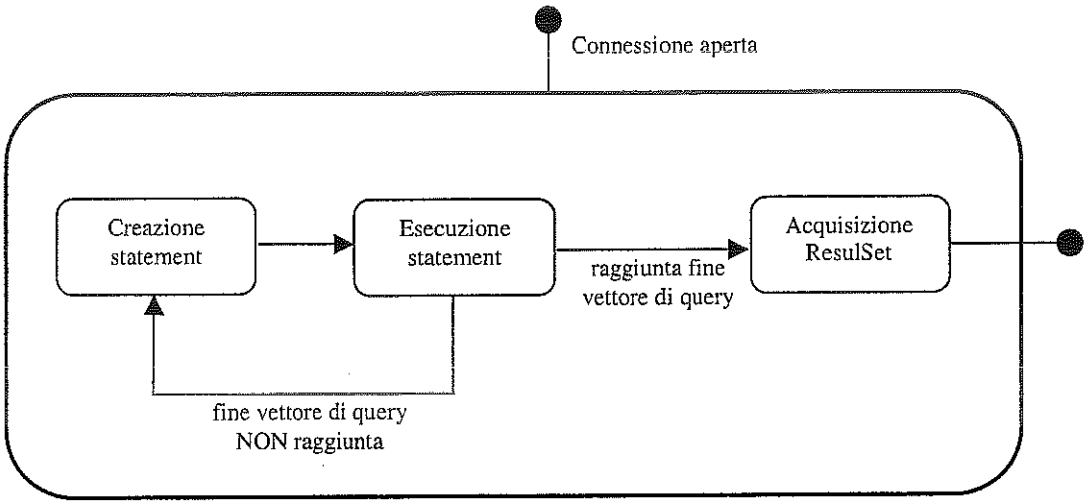

Figura 15d Diagramma dei sottostati di ESECUZIONE QUERY

In figura 16 viene presentata la gerarchia e la struttura degli attori coinvolti nel meccanismo di ricerca.

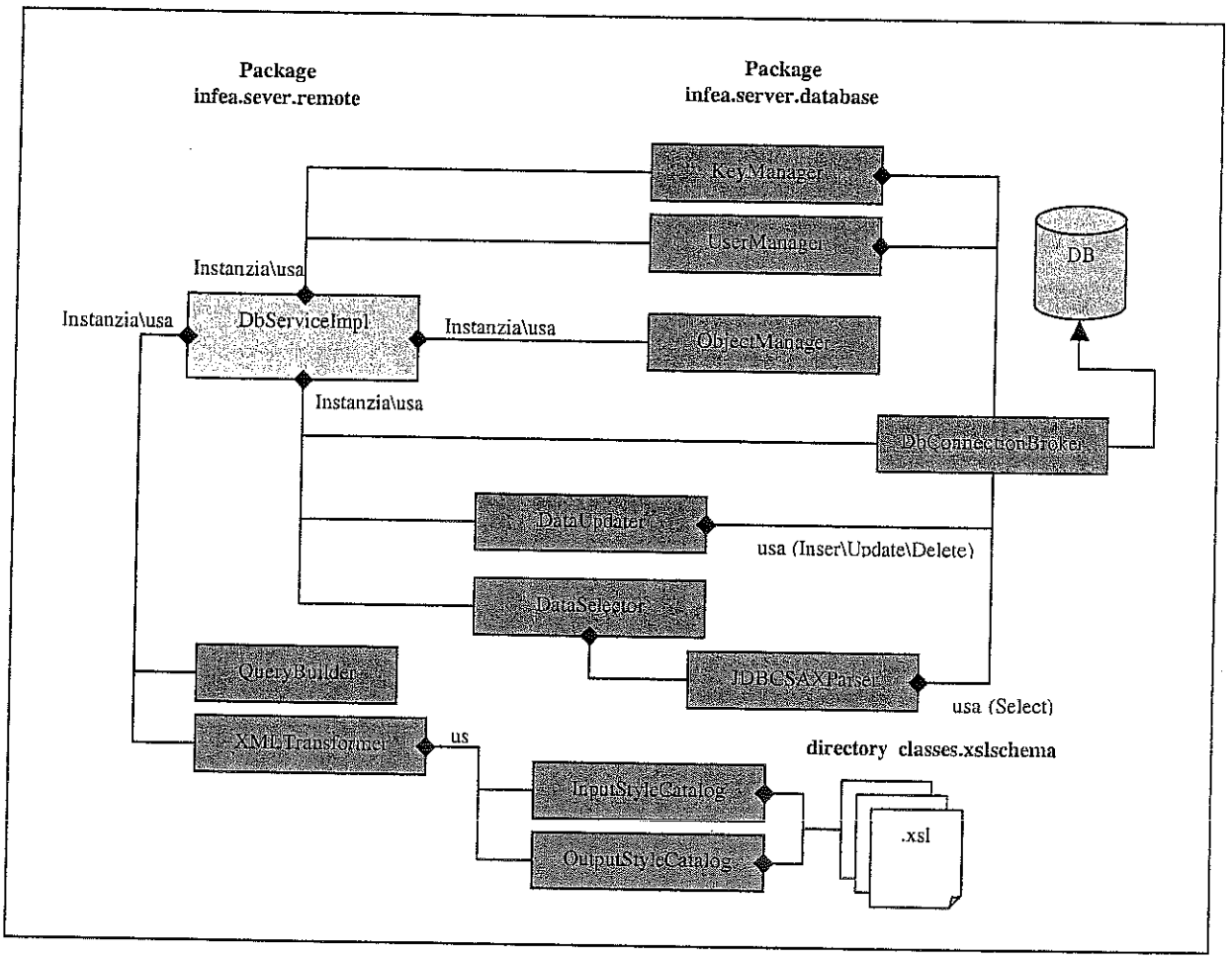

Figura 16 Gerarchia e struttura degli attori coinvolti nel meccanismo di ricerca.

4.2.5. Apertura di un modulo di immissione/modifica dati.

Il meccanismo di apertura di un modulo per immettere o per modificare i dati scaturisce nel momento in cui l'utente preme il pulsante "*Nuovo*", o nel momento in cui, selezionato un oggetto del result set, preme il pulsante "*Apri*".

Nel primo caso l'azione dell'utente viene catturata dalla classe *LB*Modulo*Search* che attraverso il metodo *doNew()* demanda l'apertura di un nuovo frame alla classe *LBCommands*: per mezzo del metodo *open*Modulo*()*, viene aperto un frame interno *LBFrame* a cui viene passato come argomento

il componente *LB*Modulo.

Nel caso di apertura di un modulo già esistente nel database, l'azione dell'utente viene catturata sempre dalla classe *LB*Modulo*Search* che attraverso il metodo *doNew(Id*Modulo*)* demanda alla classe *LBCommands* l'apertura del frame. L'identificativo *Id*Modulo è un dato appartenente al result set di ricerca, è quindi facilmente reperibile dalla classe *LB*Modulo*Search* (è suffiente una semplice lettura della tabella contenente il result set) e corrisponde alla chiave con cui il modulo è stato memorizzato nel database. Il frame che comparrà all'utente non sarà un frame vuoto ma conterrà le informazioni riguardanti il particolare modulo di cui l'utente ha richiesto l'apertura.

Attraverso il metodo *open*Modulo*(Id*Modulo*)*, *LBCommands* costruisce il vettore di richiesta di caricamento che contiene la chiave *Id*Modulo, il tipo di operazione (*UPLOAD_REQUEST*), il tag associato a tale operazione (*SINGOLO*MODULO), l'Id della sessione e il nome del database. *LBCommands* crea poi l'oggetto *LB*Modulo lasciando allo stesso il compito di richiedere al server tutte le informazioni necessarie al suo caricamento. Tale funzione in realtà viene svolta dalla superclasse di *LB*Modulo, *LBDefaultDataView*, che tramite il metodo *setXmlSource(...)* richiama la *SendQuery(...)* di *LBService* e si pone in attesa del risultato. In seguito alla richiesta di *SendQuery(...)* scattono gli stessi meccanismi coinvolti in una richiesta di ricerca, con qualche differenza che illustreremo di seguito rimandando per una descrizione più dettagliata alla sezione precedente e alle figure 15a/b/c/d.

DBService_Impl dopo aver eseguito i dovuti controlli sull'utente e sulla disponibilità degli oggetti coinvolti nella richiesta, esegue un lock su tali oggetti (metodo *lockObject(...)* della classe *objectManager*) affinchè una eventuale modifica degli stessi non crei inconsistenze nei dati. Quindi esegue la prima trasformazione della struttura dei dati, attraverso il metodo *applyInputStyles(...)* recuperando dalla classe *Input_StyleCatalog* il foglio di stile associato al tag che accompagna la richiesta di caricamento, *SelezionaSingolo*Modulo*.xsl*.

Si procede come nel caso della ricerca, attraverso l'azione di *QueryBuilder* otteniamo la risposta dal server a cui viene sottoposta una seconda trasformazione. Tale trasformazione avviene in questo caso in base al foglio di stile *Singolo*Modulo*.xsl* recuperato dalla classe *Output_StyleCatalog*.

Quando l'esecuzione ritorna a *LBDefaultDataView* i dati arrivati dal server vengono passati ad ogni

classe che realizza i pannelli di *LB*Modulo affinchè queste possano visualizzare graficamente i valori che costituiscono l'intero modulo richiesto dall'utente, metodo *setXmlData(...)*.

Il Modulo è infine visualizzato all'interno di un frame attraverso il metodo *openInternalFrame(...)* di *LBCommands*.

*LB*Modulo è quindi la classe che implementa il modulo di immissione e di modifica dei dati. Tale classe deriva da una gerarchia di classi astratte e concrete che si trovano nel package infea.client.common, e sono denominate: *LBMultiplePanelView*, *LBTreeView*, *LBDefaultDataView*[10]. Queste classi definiscono lo stile di presentazione dell'interfaccia, comune ad ogni modulo, e, in modo trasparente, gestiscono tutte le operazioni di comunicazione con il lato server di Infea. La specificità di ogni modulo funzionale si ottiene aggiungendo le classi che implementano l'interfaccia[11] *LBCardPanel*.

Le relazioni tra le varie classi sono mostrate in figura 17.

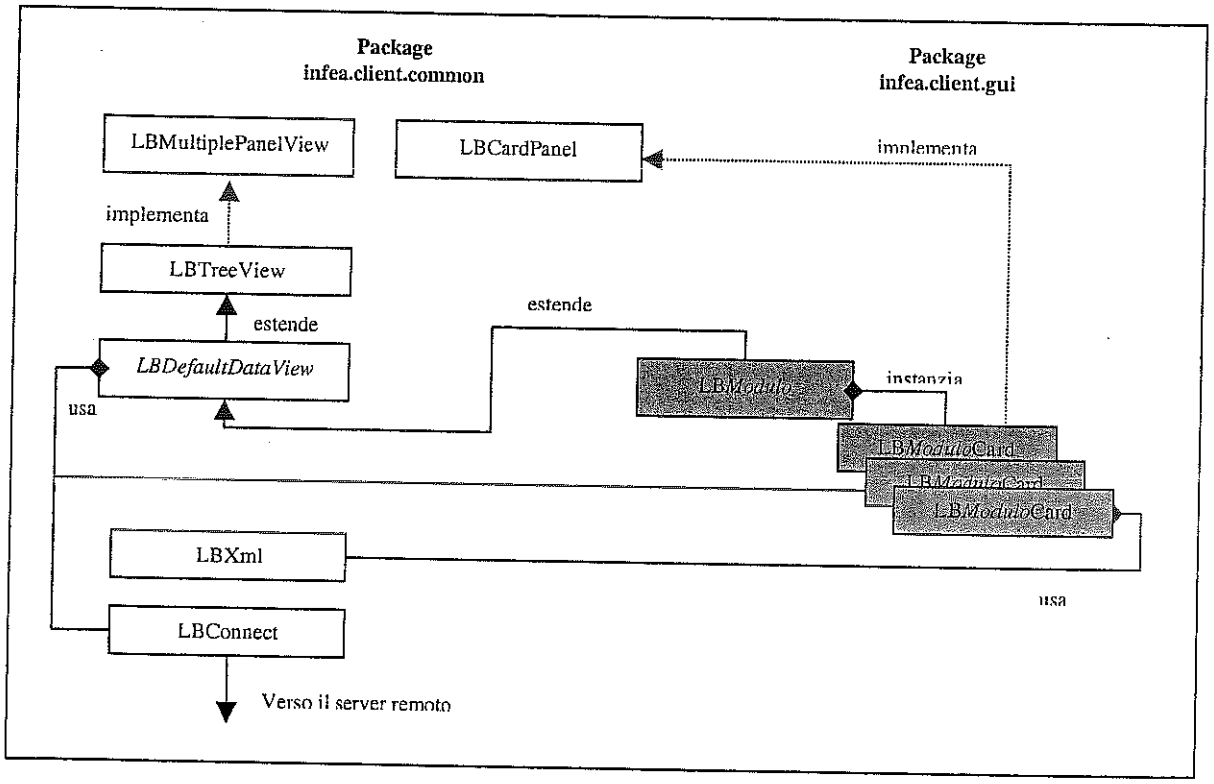

Figura 17 Gerarchia e struttura dei moduli di immissione e modifica.

[10] I sorgenti di LBMultiplePanelView, LBTreeView, LBDefaultDataView assieme a LBCardPanel appartengono al progetto Common in Z/Makes/infea/client/common.

[11] In questo contesto, il temine interfaccia è utilizzato nell'accezione Java, per riferire una componente di tale linguaggio.

La realizzazione di ogni modulo funzionale si basa su due interfacce Java che definiscono il framework di presentazione e quello delle operazioni. Queste interfacce impongono che ogni modulo di immissione e di modifica sia composto da più pannelli, e che ogni pannello implementi le operazioni, con le quali leggere ed aggiornare i dati di sua pertinenza. Possiamo riassumere dicendo che le classi astratte modellano il comportamento operazionale (*factory method pattern*), mentre gli aspetti grafici vengono deferiti alle sottoclassi concrete.

Vediamo in dettaglio.

LBTreeView estende JPanel e implementa *LBMultiplePanelView*. Tramite JSplitPane, suddivide graficamente il pannello in due sezioni verticali: la sezione di sinistra è una "sezione di scorrimento", consta di un albero JTree i cui nodi rappresentano ognuno un pannello di cui è costituita la sezione di destra, e serve appunto per scorrere tale sezione. La sezione di destra, "sezione di immissione", è costituita da pannelli in cui l'utente è in grado di inserire i dati e da alcuni bottoni. In *LBTreeView* vengono creati e implementate le funzionalità di due bottoni, (quelli che permettono di scorrere sequenzialmente i pannelli) dando la possibilità di aggiungerne altri attraverso il metodo *addButton(...)*. I pannelli costituenti la parte sinistra sono oggetti della classe *LBCardPanel* e vengono aggiunti tramite il metodo *add(...)*.

LBDefaultDataView è la classe astratta che estende *LBTreeView*: crea e dà funzionalità a tre nuovi bottoni, "Nuovo", "Elimina", "Salva" e implementa una serie di metodi che agiranno nel momento in cui l'utente farà una richiesta di nuovo inserimento, di eliminazione o di salvataggio dati.

*LB*Modulo è la classe concreta che estende *LBDefaultDataView*. In questa classe vengono creati gli oggetti che implementano l'interfaccia *LBCardPanel*. Tali oggetti costituiscono ognuno un pannello che compone la "sezione di immissione" del modulo e vengono caricati attraverso il metodo *addCards()* che richiama il metodo *add(...)* di *LBTreeView*. Vengono inoltre definiti l'icona e il titolo che devono comparire nella barra del titolo (*getIcon()*, *getNewTitle()*, *getTitle()*), e i tag usati per identificare il tipo di operazione richiesta (*getDeleteTag()*, *getInsertTag()*, *getModifyTag()*).

L'interfaccia *LBCardPanel* definisce i metodi che vengono utilizzati al momento del salvataggio o cancellazione dei dati e che vengono implementati da ogni classe che realizza ognuno un pannello della sezione di immissione.

4.2.6. Salvataggio di un inserimento.

Quando l'utente apre un nuovo modulo, premendo il pulsante "*Nuovo*", inserisce i dati e preme il pulsante "*Salva*", dà avvio al meccanismo di immissione nel database delle informazioni inserite.
L'azione sul pulsante provoca l'esecuzione del metodo *doSave()* della classe *LBDefaultDataView*. Vediamone il funzionamento: dapprima viene fatto un controllo sulla validità dei dati (che dati obbligatori siano stati inseriti, che campi numerici contengano effettivamente numeri, etc.). A tale scopo viene richiamato il metodo *doValidate()* (classe *LBCardPanel*) che troviamo implementato in ogni pannello costituente "la sezione di immissione" e tramite cui viene controllato ogni inserimento critico del pannello specifico.
Superati positivamente tutti i controlli, *LBDefaultDataView* richiede ad ogni pannello (metodo *doGetXmlData(...)*) di strutturare i dati inseriti dall'utente in formato XML (metodo *getXmlData(...)* di *LBCardPanel*); quindi tali dati vengono raccolti e concatenati, viene aggiunta l'informazione sul tipo di operazione, *INSERT_REQUEST*, e un tag opportuno, *INSERISCI*MODULO, ottenendo la richiesta di inserimento. *LBDefaultDataView* invia infine tale richiesta al server attraverso il metodo *SendQuery(...)* di *DBService*, e si pone in attesa della risposta. Il metodo *SendQuery(...)*, invocato per una richiesta di inserimento, agisce in modo leggermente diverso da quanto già detto per gli altri tipi di richiesta: *DBService* deferisce l'esecuzione della query alla classe *DBService_Impl* che compie una prima trasformazione dei dati attraverso il foglio di stile legato al tag di inserimento, *Inserisci*Modulo.*xsl*. Tale foglio xsl oltre ad apportare delle trasformazioni alla struttura dei dati, provvede a creare una chiave per ogni oggetto che dovrà essere identificato univocamente nel database.
Infine *DBService_Impl* richiede a *QueryBuilder* il vettore di query, e in seguito, essendo stato compiuto un inserimento e non una ricerca, *DBService_Impl* interagisce con la classe *DataUpdate*, package infea.server.database.updater, piuttosto che con la classe *DataSelection*: attraverso il metodo *doDataUpdating(...)* viene aperta una connesione e viene richiamato il metodo *executeUpdate(...)* della classe *Statement* che esegue una query SQL di INSERT. Se l'esecuzione della query va a buon fine *DBService_Impl* richiede il lock degli oggetti coinvolti e libera la

connessione, (l'unlock degli oggetti viene eseguito da *LBFrame* che richiama il metodo *addSwitch(...)* della classe *LBFrameSwitcher* di infea.client.common, e avviene nel momento in cui l'utente richiede la chiusura del modulo).

Una volta liberata la connessione l'inserimento nel database è ormai avvenuto, non resta che recuperare i dati da visualizzare all'utente. A tale scopo *DBService_Impl* usa il metodo *doDataSelection(...)* per eseguire una query di select che recupera i dati appena inseriti, (si veda sezione precedente).

Al risultato ottenuto, *DBService_Impl* applica, attraverso il metodo *applyOutputStyles(...)*, una seconda trasformazione dei dati utilizzando il foglio di stile *VerificaSingolo*Modulo.*xsl*. I dati così ristrutturati vengono trasferiti da *DBService* a *LBDefaultDataView* che li stava attendendo.

Il metodo *doSave()* di *LBDefaultDataView* prosegue memorizzando in una variabile l'Id che identifica univocamente il modulo da aprire (*setId()*) e commutando il titolo di default della barra del modulo nel valore inserito dall'utente (*setTitle()*) rimandando alla classe *LBFrame* il compito di abbreviarlo nel caso superi in lunghezza l'ampiezza del frame.

4.2.7. Salvataggio di una modifica.

Quando l'utente, premendo il pulsante "*Apri*", richiede l'apertura di un modulo già inserito nel database, ne modifica qualche dato o ne inserisce di nuovi e infine preme il pulsante "*Salva*" scatta il meccanismo di modifica del modulo.

Come nel caso di salvataggio di un inserimento, l'azione sul pulsante provoca l'esecuzione del metodo *doSave()* della classe *LBDefaultDataView*. Il meccanismo di salvataggio dati in seguito a una modifica è infatti molto simile al meccanismo che scaturisce in fase di inserimento perciò di seguito vengono descritte solo le differenze. Una prima differenza consiste nella costruzione della richiesta, il tipo dell'operazione in questo caso è *UPDATE_REQUEST*, mentre il tag associato è *MODIFICA*MODULO.

Ne consegue che le trasformazioni parziali avvengano grazie a fogli di stile differenti, la trasformazione in andata è provocata da *Modifica*Modulo.*xsl* e quella di ritorno da

*VerificaModifica*Modulo.*xsl*.

Di seguito verranno evidenziate altre differenze, l'una riguarda il meccanismo di bloccaggio dei dati e l'altra il processo per garantire la consistenza dei dati visualizzati con quelli effettivamente memorizzati nel database.

Il meccanismo di bloccaggio dei dati è iniziato dalla classe *DBService_Impl* che, prima di applicare la trasformazione di andata dei dati, esegue il lock degli oggetti coinvolti nella modifica richiamando il metodo *lockObject(...)* di *objectManager*, mentre l'unlock viene eseguita da *LBFrame* nel momento in cui l'utente chiude il frame che contiene il modulo.

Il processo di consistenza dei dati graficamente visualizzati consiste nel mantenere i dati visualizzati nel ResulSet allineati con i valori effettivamente presenti nel database, facciamo un esempio: dal result set del modulo di ricerca di enti l'utente seleziona la riga relativa all'ente dal titolo X e preme il pulsante "Apri". Lavora sul modulo aperto apportando una modifica al titolo, da X a Y, salva e chiude. Se il salvataggio dei dati avviene correttamente, nel database non esiste più l'ente di nome X ma esiste l'ente di nome Y. Quando l'utente ritorna al modulo di ricerca , alla riga corrispondente al modulo modificato dovrà trovare il titolo Y e non X. Tale processo viene svolto dal metodo *doSave()* della classe *LBDefaultDataView*: quando la classe ottiene da *DBService* il risultato della query, oltre a eseguire le stesse azioni previste per l'inserimento, richiama il metodo *updateAction(...)* di *LB*Moduli*Search* passandogli il risultato del metodo *getLeadingValues()* della classe *LB*Modulo. I metodi *getLeadingValues()* e *updateAction(...)* si occupano appunto di aggiornare il ResulSet.

Col metodo *getLeadingValues()*, *LB*Modulo raccoglie in un vettore i valori di alcuni campi (quelli visualizzati nel ResulSet) che risultano esistenti nel modulo al momento del salvataggio. Col metodo *updateAction(...)*, *LB*Moduli*Search* scorre il vettore ricevuto durante il salvataggio dei dati e inserisce i valori nelle opportune posizioni del ResulSet.

Un'ultima precisazione da fare sui meccanismi di salvataggio è la seguente: se un modulo viene aperto, salvato e richiuso senza avervi apportato modifiche, nessuna operazione di update viene eseguita: infatti al momento del salvataggio, attraverso il metodo *doFixData()*, tutti i valori del modulo vengono memorizzati in variabili, *old*valore, in modo che una richiesta di modifica venga

formulata soltanto se esiste almeno un campo il cui valore al momento del salvataggio è diverso dal rispettivo *old*valore e quindi solo se esiste almeno un campo che abbia subito una modifica (tali controlli avvengono durante l'esecuzione del metodo *getXmlData()* di ogni *LBCardPanel*).

In Figura 18 è illustrato il diagramma degli stati del processo di salvataggio in seguito ad una modifica. La Figura 18a rappresenta il diagramma dei sottostati dello stato rappresentato dal rettangolo azzurro.

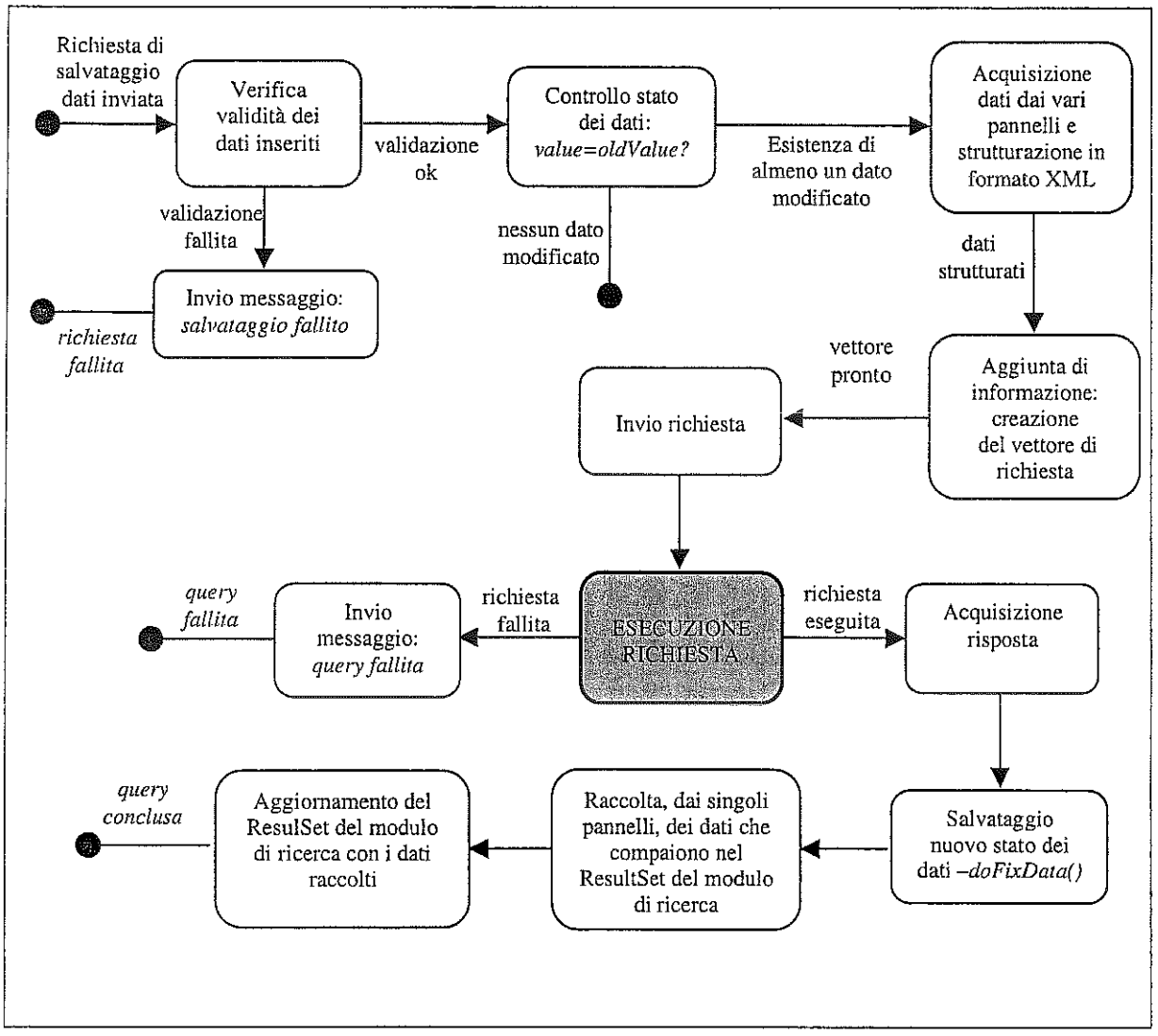

Figura 18 Diagramma degli stati per il processo di salvataggio modifiche.

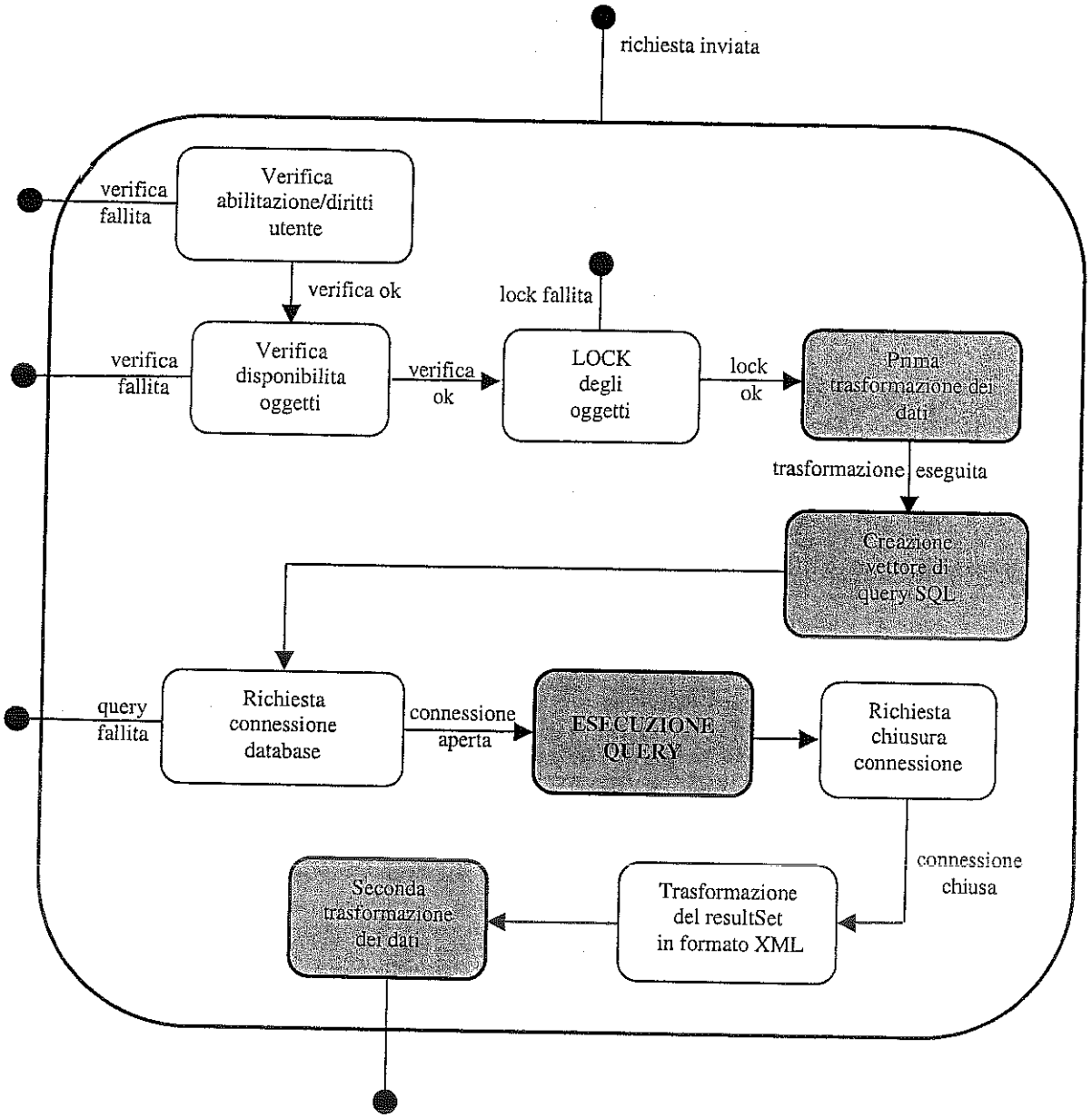

Figura 18a Diagramma dei sottostati di ESECUZIONE RICHIESTA SALVATAGGIO

Per i dettagli degli stati rappresentati in azzurro si rimanda alle figure 15b/c/d.

4.2.8. Eliminazione di un modulo.

L'utente ha la possibilità di cancellare un modulo semplicemente premendo il pulsante "*Elimina*". L'azione sul pulsante provoca l'esecuzione del metodo *doDelete()* della classe *LBDefaultDataView*. Essendo la cancellazione un'operazione critica viene chiesto all'utente una conferma, tramite

l'apertura di una finestra di dialogo. Quindi viene creato il vettore di richiesta associando all'Id che identifica il modulo da cancellare, il tipo di operazione (*DELETE_REQUEST*), il tag associato all'operazione (*CANCELLA*MODULO), l'Id della sessione e il nome del database.

Tale richiesta viene poi inviata al server attraverso il già visto *SendQuery(...)* di *DBService* che compie le stesse azioni scaturite da una richiesta di modifica tranne per il fatto che non viene lanciata una query di selezione/ricerca per recuperare i dati da visualizzare all'utente e non viene creata nessuna trasformazione di ritorno della struttura dei dati per il semplice motivo che la cancellazione non richiede alcun ritorno di informazione.

Per quanto riguarda la trasformazione di andata viene effettuata grazie al foglio di stile *Cancella*Modulo.*xsl*.

A questo punto il *doDelete()* di *LBDefaultDataView* richiama il metodo *deleteAction(...)* di *LB*Moduli*Search* che si occupa di cancellare dal result set la riga contenente le informazioni del modulo appena cancellato.

4.2.9. Accesso al sistema via browser.

Le richieste che arrivano da client basati sul browser sono gestite da moduli composti di tre servlet che realizzano le funzioni di ricerca e navigazione. I servlet con nome *Filtro*Modulo*Servlet* generano le pagine hmtl contenenti i form per esprimere le condizioni di ricerca. I servlet con nome *Elenco*Modulo*Servlet* elaborano i parametri (condizioni di ricerca) inviati dal client attraverso il metodo *getParameter(...)* e presentano l'elenco delle informazioni trovate. Infine i servlet con nome *Dettaglio*Modulo*Servlet* producono il dettaglio delle informazioni per ogni oggetto puntato dall'elenco fornito, senza però causare il lock dell'oggetto dal momento che l'utente, accedendo al sistema via browser, non è abilitato a eseguire delle modifiche quindi non sussiste pericolo di generare inconsistenze nei dati, (l'unlock dell'oggetto viene forzato attraverso il metodo *unlockObject(IdModulo)* della classe *Common*).

Sia i servlet *Elenco*Modulo*Servlet* che *Dettaglio*Modulo*Servlet* costruiscono una richiesta in formato XML, tramite i parametri ricevuti dal client, associandole il tag di identificazione dell'operazione,

rispettivamente *ELENCO*MODULO e *SINGOLO*MODULO. La inviano quindi, tramite la classe *Common*, al server RMI che agisce come nel caso la richiesta gli arrivi da un applet, (si faccia riferimento quindi a § 4.2.4. e § 4.2.5.); ricevuta la risposta dal server, la convertono nel formato adatto al tipo di client (html,wml, etc).

Tutti i servlet fanno uso di alcune classi istanziate in fase di startup dal server Web. In particolare la classe *Common* contiene metodi per l'accesso e la comunicazione con il server RMI, e la classe *XSLServlet* applica le operazioni di trasformazione in base ai fogli di stile contenuti nella directory *xslschema*.

In figura 19 è illustrata la gerarchia e la struttura dei servlet.

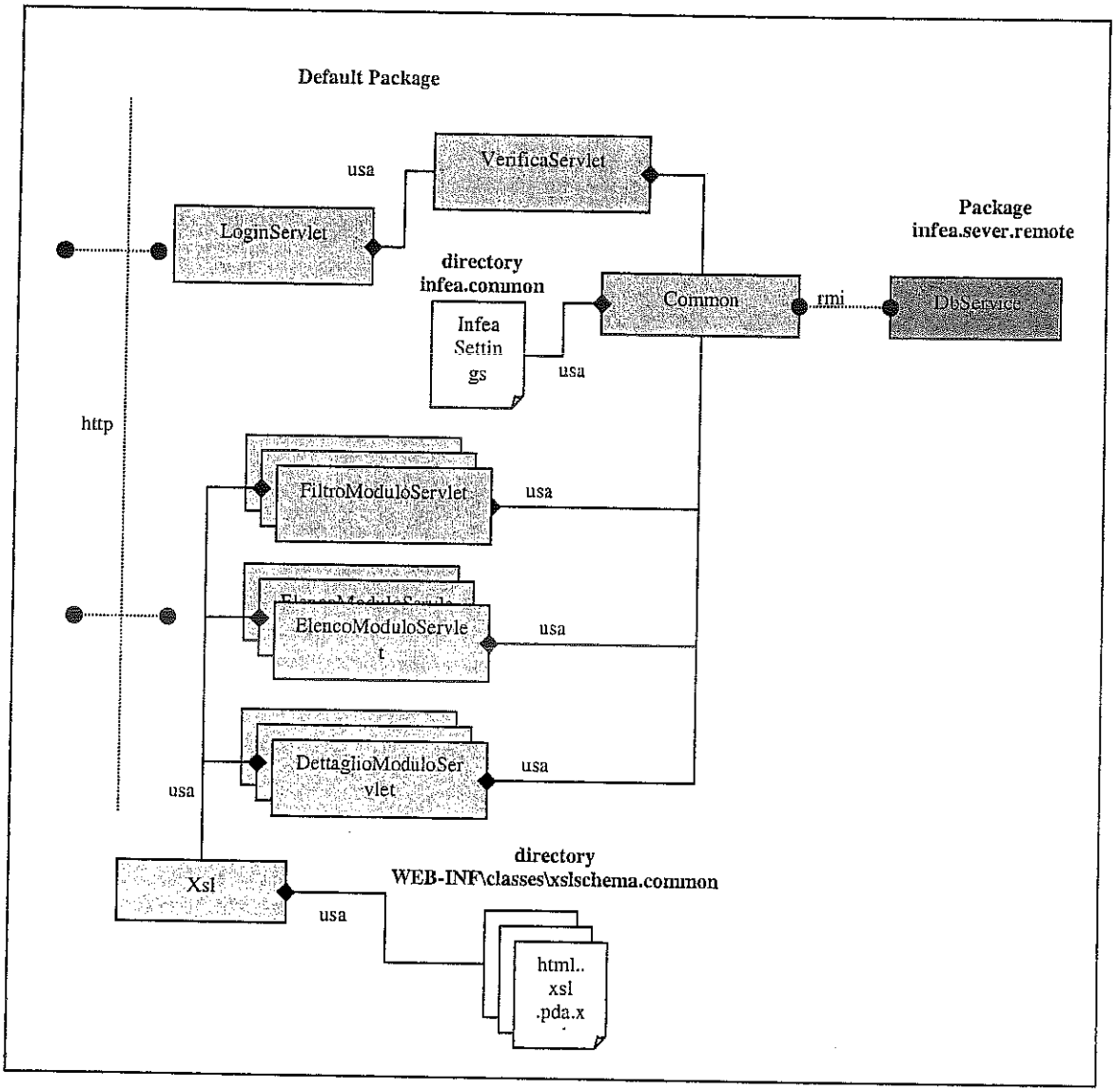

Figura 19 Gerarchia e struttura dei servlet.

5. Conclusioni.

Nel presente lavoro sono state analizzate le caratteristiche di un sistema informativo basato sul Web per la pubblica amministrazione, il sitema INFEA. Sono stati inizialmente discussi i vantaggi dell'uso della nuova tecnologia rispetto all'implementazione di un sistema informativo tradizionale all'interno della realtà aziendale e pubblica. Sono stati quindi presentati gli strumenti tecnologici utilizzati per la realizzazione del sistema INFEA, è stata data una descrizione dell'architettura software e della progettazione delle interfacce, e una dettagliata analisi dei meccanismi implementativi che scaturiscono durante l'utilizzo del sistema.

Bibliografia

[Aloia98] N.Aloia, C.Concordia **Accesso a basi di dati via Web** Proc. of SEBD '98 "Sistemi evoluti per basi di dati", Ancona giugno 1998, (pp. 3-18)

[Aloia99] N.Aloia, C.Concordia, V. Miori, **Web Architectures for Database Access,** Proc. of WebNet '99, Honolulu, Hawaii, ottobre 1999 (pp. 1473-1475)

[Aloia00] Nicola Aloia, Cesare Concordia, Vittorio Miori. **Web Based Information Systems,** Proc. of 16th IFIP World Computer Congress 2000, Information Technology for Business Management, Bejing agosto 2000, (pp. 581-588)

[Aloia01] N.Aloia, C.Concordia, V. Miori, Francesco Furfari, **Caratteristiche, problematiche e tecnologia dei sistemi informativi basati sul Web** AICA, n°2 maggio-settembre 2001 (pp.95-108).