Alma Mater Studiorum
Università di Bologna

FACOLTÀ DI SCIENZE MATEMATICHE, FISICHE E NATURALI
Corso di Laurea in Informatica

SERENA: UN FRAMEWORK DI SVILUPPO DI APPLICAZIONI WEB A PARTIRE DA ONTOLOGIE

Tesi di Laurea in Linguaggi e Strutture

Relatore:
Chiar.mo Prof.
CLAUDIO SACERDOTI COEN

Presentata da:
VINCENZO CARNAZZO

Sessione II
Anno Accademico 2009-2010

A Maura per ciò che è
A Sofia per ciò che sarà

Introduzione

L'avvento del *cloud computing* sta portando i software da realtà locali (il singolo computer dell'utente) a realtà sempre più remote, distribuite e interconnesse. Contemporaneamente, nella percezione dell'utente medio, il computer sta sempre di più trasformandosi da semplice "raccoglitore di dati" ad "assistente" che, a partire da dati noti (perché immessi dall'utente o perché raccolti in rete), fornisce all'utente supporto e suggerimenti relativi al suo lavoro.

In questo scenario la Cooperativa Anastasis, azienda che da vent'anni si occupa principalmente di sviluppo di software per l'integrazione sociale e lavorativa di persone svantaggiate, ha iniziato un percorso per poter essa stessa sviluppare applicazioni web supportate da un sistema esperto. A tale scopo è stato creato Serena, un framework di sviluppo per applicazioni web, applicazioni che poi vengono eseguite all'interno del Serena Application Server.

Il sistema Serena (cioé il framework e l'*application server*) è l'oggetto di questa tesi: scopo della tesi è analizzare lo scenario (in termini di prodotti esistenti) nel quale Serena è nato, analizzare il suo funzionamento e la sua struttura, mostrarne pregi e difetti e conseguentemente valutare gli sviluppi futuri più idonei per migliorarlo.

Nel capitolo 1 verrà descritto lo scenario (in termini di prodotti informatici esistenti) nel quale è stato avviato lo sviluppo del sistema Serena, cioé cosa è presente sul mercato e nella ricerca come framework di sviluppo di applicazioni web, come framework di sviluppo di ontologie, come sistemi

middleware e come *application server*.

Nel capitolo 2 verrà spiegato l'utilizzo del Framework Serena per la creazione di un'applicazione Serena di base e per la sua personalizzazione.

Nel capitolo 3 verrà mostrata la struttura del Serena Application Server e di come vengono gestite le richieste inviate alle applicazioni Serena. Nello stesso capitolo verrà descritto il *Serena Virtual Network*, che permette alle applicazioni Serena di comunicare con altre applicazioni in rete.

Nel capitolo 4 verranno descritte le modalità di test utilizzate dal team di sviluppo e come Serena garantisce il rispetto di alcuni standard di robustezza nonché delle leggi italiane sull'accessibilità e il rispetto della privacy nella Pubblica Amministrazione.

Nel capitolo 5 si tornerà ad analizzare la struttura del sistema Serena, evidenziandone pregi e difetti rispetto ad altre soluzioni esistenti.

Nel capitolo 6 verranno valutati i principali sviluppi futuri, necessari per mettere in evidenza i pregi e ridurre o annullare i difetti del sistema messi in risalto dall'analisi eseguita in questa tesi.

Nel capitolo 7 verranno infine tratte le conclusioni finali.

Prima di ciò, per inquadrare tutto, viene qui di seguito raccontata brevemente l'identità e la storia di Serena e della Cooperativa Anastasis, azienda che ha ideato, progettato e sviluppato Serena.

Cooperativa Anastasis

La Società Cooperativa Anastasis nasce a Bologna nel 1985 da un gruppo di tecnici informatici formatisi lavorativamente all'interno dell'Associazione ASPHI (Associazione per lo Sviluppo Professionale degli Handicappati in campo Informatico, che successivamente è diventata Fondazione e ha cambiato l'acronimo in Avviamento e Sviluppo di Progetti per ridurre l'Handicap mediante l'Informatica).

Fin dalla nascita la Cooperativa si pone come obiettivo la creazione di soluzioni tecnologiche per l'integrazione sociale e lavorativa di persone svan-

taggiate, nonché la formazione di tali persone all'uso delle tecnologie informatiche. Con il tempo il *core business* della Cooperativa si è focalizzato sui software didattici/riabilitativi e i software compensativi per persone con disturbi specifici dell'apprendimento (dislessia, disgrafia, discalculia ecc.), pur non abbandonando mai del tutto l'attenzione verso altri tipi di svantaggi (cecità, sordità ecc.).

Negli ultimi anni la Cooperativa si è aperta al mondo web, realizzando portali accessibili, dando supporto alle Pubbliche Amministrazioni per l'adattamento dei siti istituzionali ai fini della Legge Stanca n.4/2004 e, più recentemente, sviluppando applicazioni web sofisticate, rivolte soprattutto al mondo delle Pubbliche Amministrazioni e della Sanità, nell'ottica più ampia di creazione di servizi e proposte nel campo del *welfare-mix*, con partnership con Università, Associazioni, Cooperative Sociali e tutti quei soggetti istituzionali che, a vario titolo, operano nel Terzo Settore.

All'interno di quest'ultimo ambito nasce il progetto Serena descritto in questa tesi, la cui storia è descritta brevemente nella sezione seguente.

Storia di Serena

Serena nasce nel 2005 come progetto nell'ambito del programma regionale di ricerca industriale e sviluppo precompetitivo della Regione Emilia Romagna, progetto di cui è titolare la Cooperativa Anastasis. All'interno del progetto collaborano il Dipartimento di Elettronica, Informatica e Sistemistica (DEIS) della Facoltà di Ingegneria dell'Università di Bologna, il Dipartimento di Scienze della Formazione di Bologna e il Dipartimento di Scienze Sociali, Cognitive e Quantitative dell'Università degli Studi di Modena e Reggio Emilia.

L'obiettivo principale del progetto è la realizzazione di una infrastruttura integrata per la gestione dell'intero processo di "presa in carico" di utenti in situazioni di disagio, processo che solitamente comprende una fase di *screening*, una di rieducazione e una di formazione. Il compito iniziale del sistema

Serena era dunque quello di permettere l'inter-operabilità di differenti strumenti riabilitativi ed educativi (alcuni anche *legacy*), mantenendo traccia di tutta la storia clinica degli assistiti. Per tale motivo originariamente "SERENA" era l'acronimo di Sistema tErritoriale per Riabiltazione ed Educazione - Network di Accompagnamento.

Con il tempo il sistema Serena si è evoluto tanto da perdere la connotazione stretta di aggregatore di dati (e quindi perdendo anche il corrispondente acronimo), diventando al tempo stesso un framework generico di sviluppo di applicazioni web, ospitate all'interno di un apposito *application server*.

Negli anni con Serena sono state sviluppate sia applicazioni strettamente inerenti all'obiettivo iniziale (come la *Cartella Clinica Territoriale* del Polo Multifunzionale Corte Roncati di Bologna o la *Cartella per la Neuropsichiatria Infantile* della ULSS 3 di Bassano del Grappa), sia applicazioni di supporto alla diagnosi tramite un sistema esperto (*AD-DA*, Assistente alla Diagnosi dei Disturbi dell'Apprendimento), sia aggregatori di dati da software riabilitativi *legacy* (*Ri-Di*, RIabilitazione a DIstanza) sia applicazioni web più generiche (come il *Gestore Immobili* dell'AUSL di Bologna o il *Gestore delle Circolari* della Lega delle Cooperative della Provincia di Bologna), sia addirittura CMS (come nel caso del Portale della Lega delle Cooperative di Bologna o il portale La Scienza nei Musei della Fondazione IBM).

Lo sviluppo di Serena è stato perseguito dal team di sviluppo della Cooperativa Anastasis, con il contributo di alcuni tesisti della Facoltà di Ingegneria dell'Università di Bologna. La stesura di questa tesi coincide con il rilascio della versione 1.5 di Serena.

L'autore di questa tesi lavora all'interno della Cooperativa Anastasis dal 2005 ed è entrato all'interno del processo di creazione, progettazione e sviluppo di Serena dalla sua nascita, intervenendo su più livelli (progettuale e implementativo) e su più ambiti, soprattutto sulle parti relative alla comunicazione in rete delle applicazioni Serena e alla *renderizzazione* grafica dei dati (mostrati nel dettaglio nel capitolo 3).

Dal 2009 è *project leader* del Progetto Serena.

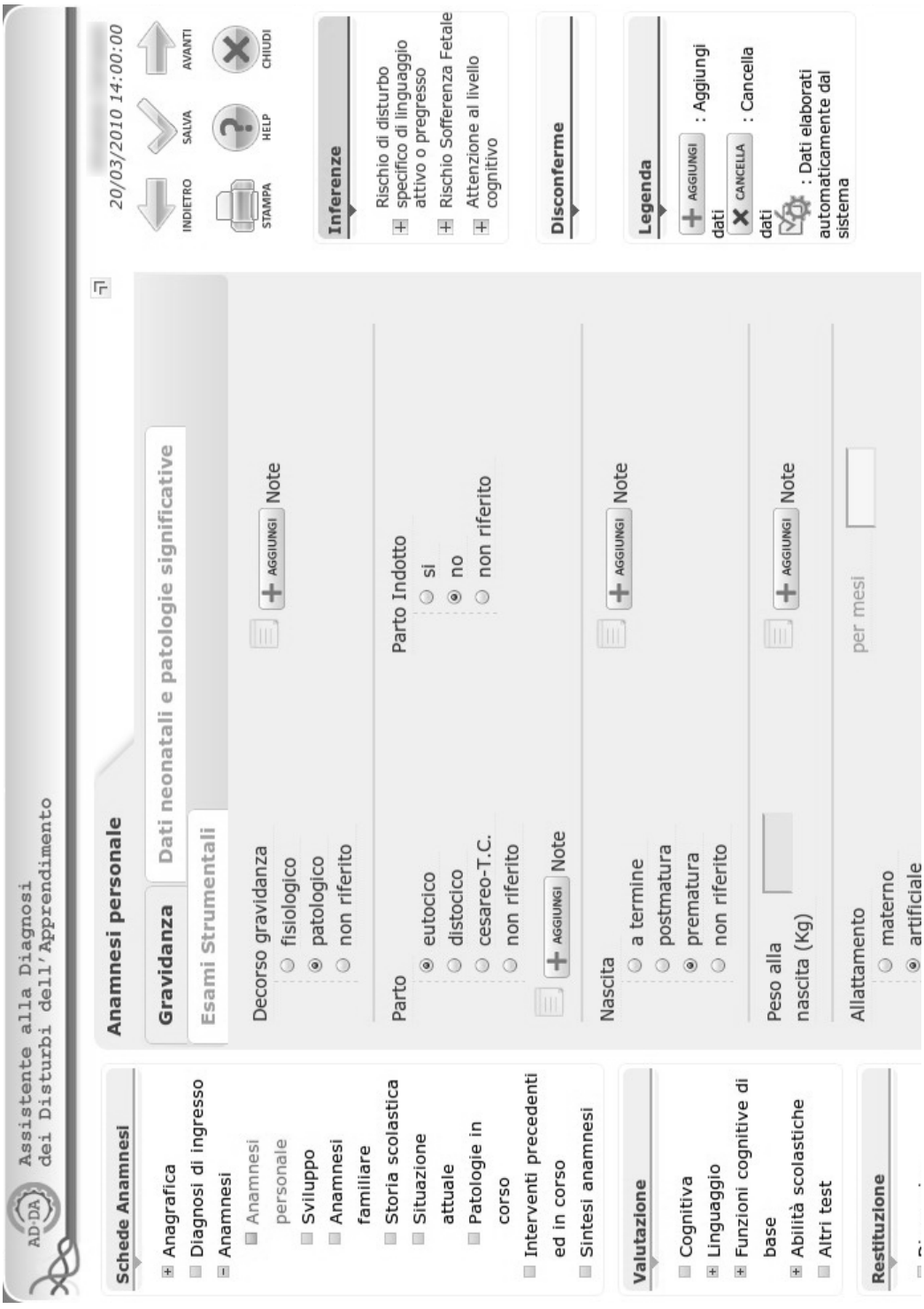

Figura 1: Schermata di esempio di AD-DA con anamnesi personale e alcune inferenze del sistema esperto

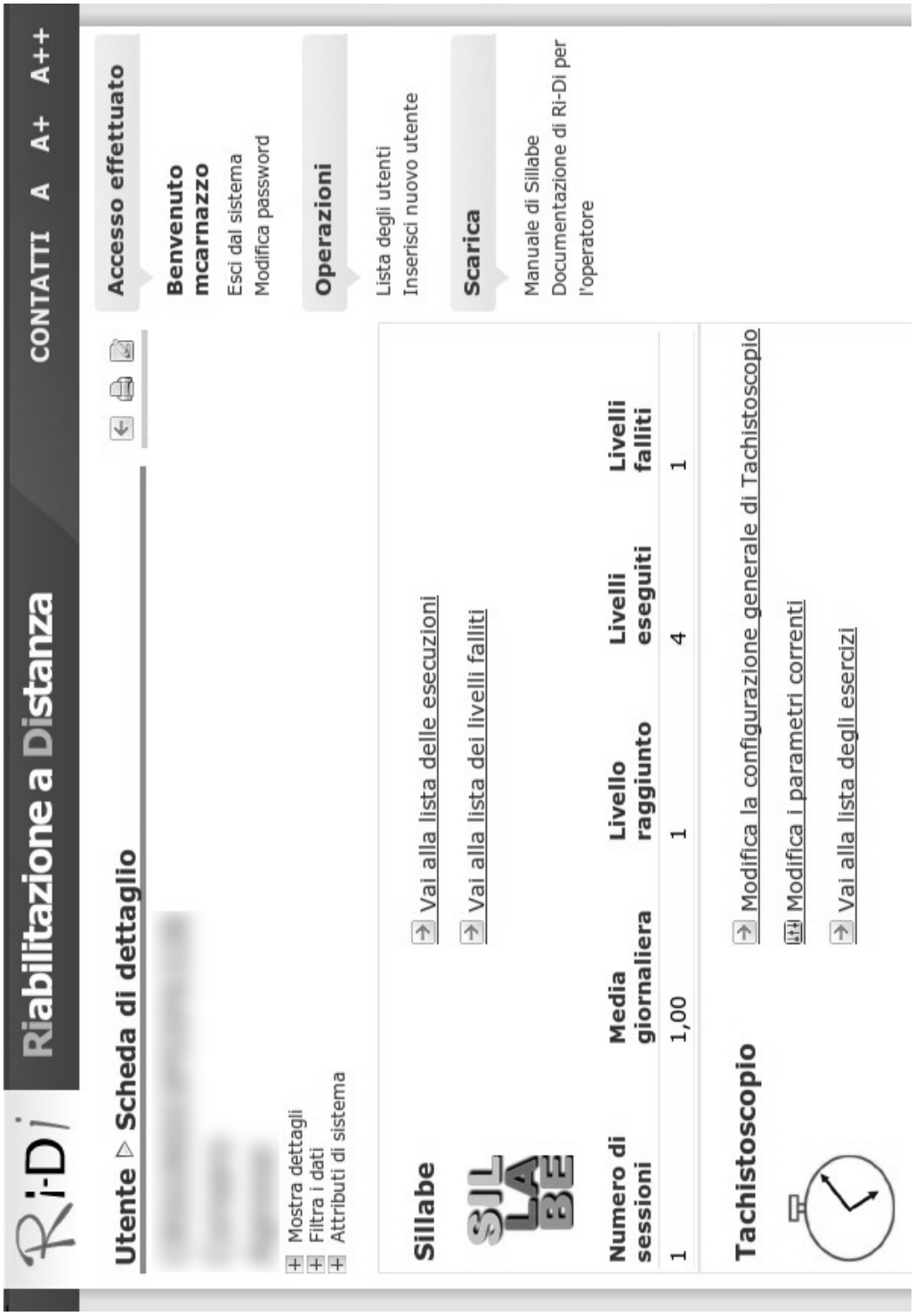

Figura 2: Schermata di esempio di Ri-Di, con il dettaglio di un utente seguito

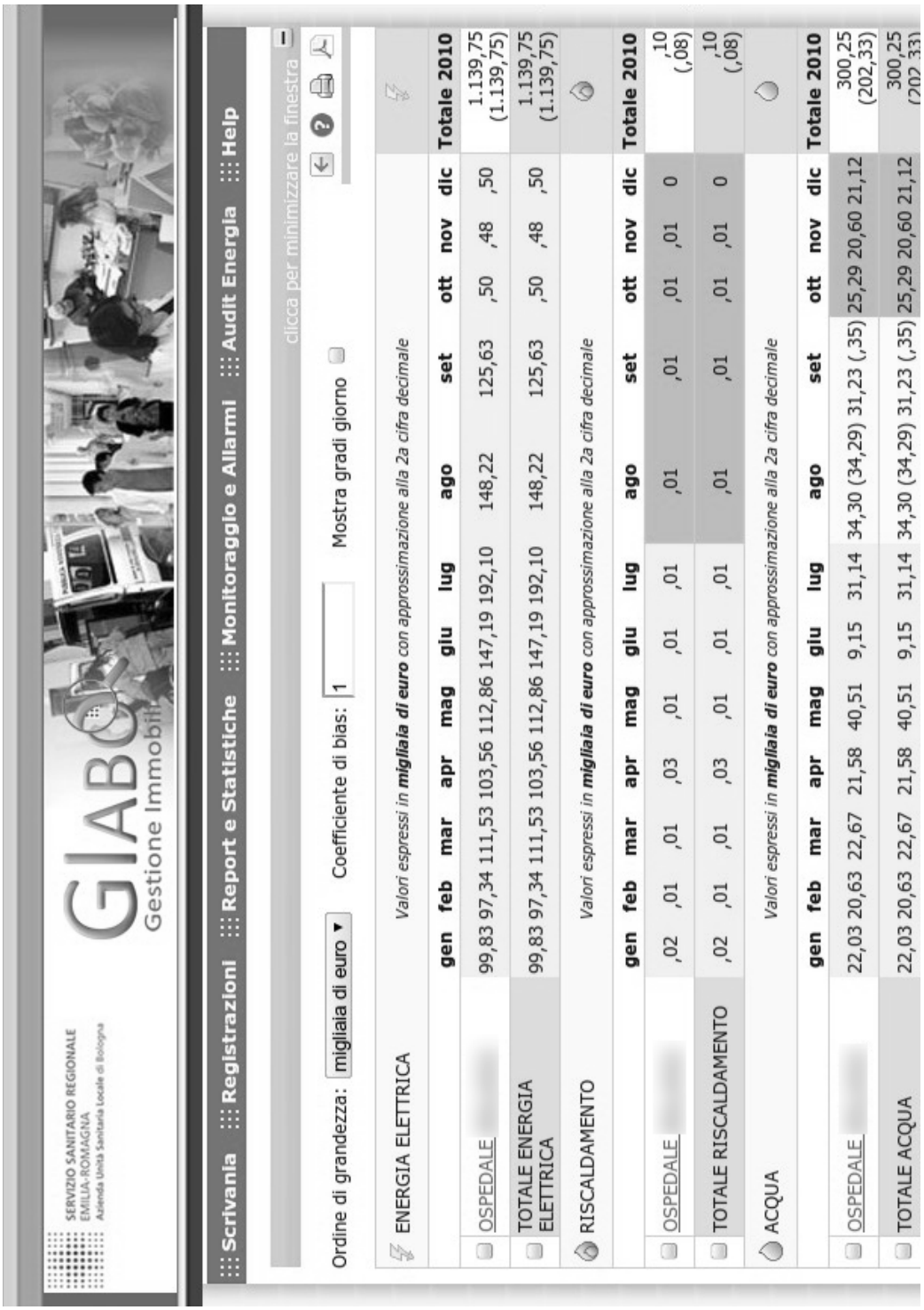

Figura 3: Schermata di esempio di Giabo, Gestore Immobili dell'AUSL di Bologna

Figura 4: Schermata della home page del portale di Legacoop Bologna

Figura 5: Schermata della home page del portale La Scienza nei Musei

Indice

Elenco delle figure

Capitolo 1

STATO DELL'ARTE

Prima di mostrare il Framework Serena, va chiarito il contesto in cui è nata l'idea e si è sviluppato il progetto. In questo capitolo verrà dunque mostrato brevemente lo stato dell'arte in termini di framework di sviluppo di applicazioni web e di applicazioni basate su ontologie, nonché su sistemi *middleware* e *application server*.

1.1 Framework per sviluppo di applicazioni web

Da qualche anno il mondo dell'informatica si sta pian piano spostando interamente sul web, dove ormai non si trovano soltanto siti di tipo consultativo o di *editing* collaborativo (come i *wiki* e i CMS), bensì anche delle vere e proprie applicazioni, nell'ottica più ampia di *cloud computing*: un trasferimento di dati e applicazioni dai personal computer locali a server online, con il vantaggio di non dover più gestire aggiornamenti e backup e di poter accedere alle proprie applicazioni in qualunque postazione che ha accesso a Internet. Di conseguenza sono fioriti una serie di framework specifici per lo sviluppo di applicazioni web. Tra questi i più significativi nella sfera dell'open

source[1] sono:

- *Django*. Usa il linguaggio di programmazione Python ed è orientato principalmente allo sviluppo di siti di notizie.
- *Ruby on Rails* (RoR). Usa il linguaggio Ruby e come principale caratteristica ha la facilità di utilizzo, dovuta alle poche righe di codice necessarie allo sviluppo e alla poca configurazione necessaria (purché si rispettino determinate convenzioni nel creare oggetti, database ecc.).
- *MonoRail*. Ispirato a Ruby on Rails, usa il linguaggio Mono, versione open compatibile con .NET.
- *Flex*. Usa il linguaggio Actionscript per la logica e il dialetto XML MXML per la grafica e crea oggetti SWF (quindi fruibili attraverso l'Adobe Flash Player). Crea risultati con una grafica accattivamente, al costo di avere un linguaggio per la parte logica non particolarmente "robusto".
- *JBoss*. Usa il linguaggio Java e fornisce un'ampia gamma di strumenti per molteplici ambiti di sviluppo. Si tornerà su questa soluzione più avanti in questo stesso capitolo.

Dovendo sviluppare applicazioni web di una certa complessità (ad esempio per gestire cartelle cliniche informatizzate) e aperte (cioé slegate da soluzioni proprietarie e *closed source*), si è deciso di orientarsi verso il linguaggio Java e adottare Eclipse come IDE di partenza su cui costruire il Framework Serena.

In alcuni punti, Serena è paragonabile al framework di sviluppo Grails[2], che non è stato considerato in fase di progettazione per il semplice fatto che nel 2005, anno in cui Serena è nato, Grails non era ancora stato rilasciato.

[1] Fin da subito il team di sviluppo di Serena, per ragioni etiche e di costi, si è orientato esclusivamente su alternative open source.

[2] Grails è un framework di sviluppo di applicazioni web, basato principalmente sul linguaggio Groovy, una sorta di "dialetto" di Java. Per approfondimenti si veda [RB09].

1.2 Framework per lo sviluppo di applicazioni basate su ontologie

Parallelamente al *cloud computing* è andata avanti in questi anni la creazione e lo sviluppo di standard per la rappresentazione dei dati in modo più libero rispetto ai classici modelli relazionali dei database. Tale via è stata spinta da un interesse verso il *semantic web* (cioé la possibilità di esprimere i dati presenti nel web in modo *machine understandable*) e i sistemi esperti (sistemi software progettati per emulare esperti umani con conoscenze specialistiche).

Quest'ultimo ambito è quello che più interessa lo scenario di utilizzo di un sistema come Serena, in quanto le applicazioni da sviluppare gestiscono conoscenze complesse e spesso hanno bisogno di un sistema esperto che assista specialisti sanitari nella diagnosi di determinati disturbi. Si è perciò deciso che il sistema Serena avrebbe gestito i dati attraverso l'uso di ontologie.

Esistono vari software (generalmente nati in ambienti universitari) che si occupano di modellazione di ontologie e spesso hanno a corredo una serie di API per accedere all'ontologia da applicazioni esterne. Tra queste le più famose sono Koan, SOFA e Protégé, le cui caratteristiche sono molto simili.

Purtroppo nessuna delle applicazioni esistenti (per lo meno nessuna abbastanza stabile e nota) segue l'intero processo di sviluppo, dalla modellazione dei dati alla creazione vera e propria delle applicazioni. Uno degli scopi principali di Serena è colmare questo vuoto.

Per la modellazione dei dati delle applicazioni Serena si è deciso di utilizzare Protégé, perché, a pari qualità di altri software analoghi, era già noto ad alcuni degli sviluppatori Anastasis. Si tornerà a parlare di Protégé nella sezione 2.1.2 per mostrarne brevemente il funzionamento e nella sezione 5.1 per mostrare le differenze tra modello ontologico e modello relazionale e quanto il primo è stato sfruttato nel sistema Serena.

1.3 Sistemi Middleware

Le applicazioni web create con un framework devono poggiarsi su un sistema che sappia gestire tutte le attività di contorno (interpretazione delle richieste HTTP, gestione delle transazioni, comunicazione con applicazioni esterne ecc.). A ciò serve un sistema middleware.

Poiché è stato deciso di orientarsi sul linguaggio Java, era inevitabile prendere in considerazione Java EE, descritto nella sezione seguente.

1.3.1 Java EE

Java Platform Enterprise Edition (o brevemente *Java EE*) è uno standard industriale sviluppato dalla Sun Microsystem (e successivamente acquisito da Oracle) per lo sviluppo di applicazioni *server-side* con il linguaggio di programmazione Java. Lo standard comprende una serie di componenti, API, web service e modelli di gestione e di comunicazione:

- *Java Servlet*, per rispondere a chiamate HTTP utilizzando classi Java.
- *Java API for Web Services* (JAX-WS) - che sostituisce *Java API for XML-Based RPC* (JAX-RPC) - per lo sviluppo di *web service*.
- *Java Architecture for XML Binding* (JAXB), per mappare classi Java in dati XML.
- *SOAP with Attachments API for Java* (SAAJ), per la creazione di documenti secondo il protocollo SOAP del W3C.
- *Streaming API for XML* (StAX), per leggere e scrivere documenti XML.
- *Web Service Metadata for the Java Platform, Enterprise JavaBeans* (EJB), per creare web service e/o oggetti persistenti tramite semplici oggetti Java con in più alcune annotazioni.

- *J2EE Connector Architecture* (JCA), per poter connettere e far interagire tra loro prodotti eterogenei.
- *Java Server Faces* (JSF), per sviluppare attraverso componenti precostituiti l'interfaccia grafica delle applicazioni server.
- *Java Server Pages* (JSP), un'astrazione delle Java Servlet che permette di "mischiare" codice HTML con codice Java.
- *Java Server Pages Standard Tag Library* (JSTL), un'estensione di JSP che permette di aggiungere tag personalizzati all'interno di pagine JSP, per separare la parte logica da quella grafica.
- *J2EE Management*, per la generica gestione di tutto un sistema Java EE.
- *J2EE Application Deployment*, per effettuare il deploy delle applicazioni, cioé metterle su un server e quindi renderle raggiungibili dall'esterno.
- *Java Authorization Contract for Containers*, per la gestione dei permessi di accesso alle classi.
- *Common Annotation for the Java Platform*, per aggiungere alle classi Java ulteriori funzionalità e automatismi aggiungendo ad esse delle cosiddette "annotazioni".
- *Java Message Service API* (JMS), per lo scambio di messaggi tra applicazioni all'interno di una rete.
- *Java Persistence API*, per mappare oggetti Java nei database relazionali.
- *Java Transaction API* (JTA), per la gestione di transazioni che coinvolgono più risorse.
- *JavaBeans Activation Framework* (JAF), per la gestione dei *bean*, classi Java che permettono in modo semi-automatico di gestire dati (crearli, modificarli, cancellarli e leggerli).

- *JavaMail*, per l'invio di posta elettronica

1.3.2 Application server

Nello scenario attuale, le soluzioni più robuste che supportano (interamente o parzialmente) le caratteristiche dei servizi Java EE sono:

- *JBoss*. Supporta tutto il set di servizi Java EE, attraverso una serie di componenti, ciascuno dedicato a una parte dei servizi. È la soluzione open source più nota e utilizzata in questo ambito.

- *Apache Tomcat*. È il più noto *servlet container* open source. Supporta gli standard JSP e Servlet di Java EE.

- *Jetty*. È anch'esso un *servlet container*, come Tomcat. Ha la particolarità di essere più leggero e più adatto a soluzioni *embedded*.

- *Resin*. È un Java application server simile a quelli appena elencati ma con la particolarità di saper gestire anche codice PHP.

Tra le soluzioni mostrate, sicuramente quella più completa è JBoss, anche perché corredata da tutta una famiglia di componenti che coprono tutte le esigenze, come ad esempio il framework di sviluppo *Java Seam*.

Ciò nonostante, nella fase iniziale di progettazione di Serena, nell'ormai lontano 2005, non era chiaro se tutti i componenti di Serena sarebbero stati sviluppati in Java: la Cooperativa Anastasis aveva già al suo interno alcuni componenti, sviluppati in altri contesti, che potenzialmente potevano essere riutilizzati (ad esempio un gestore della rappresentazione grafica, sviluppato in ASP). Si è perciò deciso di non legarsi interamente a un solo linguaggio e a una sola piattaforma e sono state quindi scartate soluzioni complesse come JBoss optando per lo sviluppo *servlet oriented* e quindi scegliendo come base di tutto il sistema un *servlet server*.

Serena funziona dunque su qualsiasi *application server* che supporti gli standard di *Java servlet*. È stato utilizzato con successo, ad esempio, con Apache Tomcat[3] e (in soluzioni client) con Jetty.

In conclusione, si è visto che esistono valide soluzioni come framework di applicazioni web, come framework di applicazioni basate su ontologie e come *application server*, ma niente che unifichi i tre concetti e che sia abbastanza semplice e automatizzato da permettere uno sviluppo veloce. Si è deciso dunque che il sistema Serena sarebbe stato composto da un framework di sviluppo e un *application server* creati ex novo, pur utilizzando al suo interno alcuni componenti open source pre-esistenti.

Il framework di sviluppo Serena e il Serena Application Server Serena saranno discussi rispettivamente nei capitoli 2 e 3. Lo stato dell'arte verrà ripreso nel capitolo 5 per confrontare ciò che è già presente con ciò che è stato sviluppato in Serena.

[3]Per maggiori informazioni su Apache Tomcat si veda [CLG07].

Capitolo 2

UTILIZZO DEL SERENA DEVELOPMENT FRAMEWORK

Il Serena Development Framework è un insieme di strumenti che permettono la creazione di applicazioni Serena, cioè applicazioni web che vengono eseguite in un Serena Application Server.

In questo capitolo viene mostrato l'utilizzo del Serena Development Framework: verrà presentato innanzitutto un tutorial su come creare un'applicazione Serena di base, seguita da una guida su come effettuare personalizzazioni più avanzate dei dati, dei comportamenti e della grafica dell'applicazione.

2.1 Come creare un'applicazione Serena

Verrà ora mostrato come creare una semplice applicazione Serena, che chiameremo "Pippo". Va chiarito che il Serena Development Framework è pensato per applicazioni sviluppate all'interno dell'IDE Eclipse[1]. Nel resto del capitolo si darà dunque per scontato che si lavora all'interno di Eclipse.

[1]Eclipse è un'IDE multi-piattaforma per lo sviluppo di applicazioni Java (e non solo). Per approfondire il funzionamento e l'uso di Eclipse si veda [Hol04].

Alcune delle operazioni descritte in questo capitolo verranno automatizzate dal *Serena Eclipse Plugin*, che è in fase di sviluppo durante la stesura di questa tesi e che verrà brevemente descritto nel capitolo 6 relativo agli sviluppi futuri.

2.1.1 Primi passi

Innanzitutto è necessario scaricare la versione più aggiornata del Serena Development Framework[2] da `www.sere-na.it`, previa iscrizione.

Va quindi creato in Eclipse un nuovo progetto Java chiamato Pippo, all'interno del quale va importato il file zip di Serena scaricato: verranno così inserite nel progetto una serie di cartelle, di librerie e di altri file necessari per lo sviluppo di una nuova applicazione Serena. La struttura delle cartelle così create sarà la seguente:

- `src`. Conterrà eventuali file Java specifici dell'applicazione. In principio sarà vuota.
- `lib`. Contiene le librerie necessarie a far funzionare l'applicazione Serena.
- `offline_generated_files`. Contiene alcuni file generati automaticamente dal Serena Developer Tool e che non verranno usati direttamente dall'applicazione. Il suo contenuto verrà approfondito nella sezione 2.1.4.
- `webapps`. Contiene tutti i file di configurazione e personalizzazione dell'applicazione Serena. A sua volta contiene:
  - `app`. Contiene i file Serena. A sua volta contiene:
    - `conf`. Contiene i file di configurazione. A sua volta contiene:

[2] Al momento della stesura di questa tesi la versione più aggiornata è la 1.5.

· `entities`. Contiene i *Serena Entity Bean* che descrivono i dati gestiti dall'applicazione. Il suo contenuto verrà approfondito nella sezione 2.6.

· `interfaces`. Contiene i *Serena Interface Bean* che descrivono gli elementi grafici di base dell'applicazione. Il suo contenuto verrà approfondito nella sezione 2.7.6.

· `system`. Contiene i file di configurazione di sistema dell'applicazione. Il suo contenuto verrà approfondito più avanti in questa sezione.

∗ `Javascript`. Contiene tutti i file javascript utilizzati all'interno delle pagine web dell'applicazione. Il suo contenuto non verrà approfondito [3].

`Template`. Contiene tutti i file che specificano come creare le pagine web dell'applicazione. Il suo contenuto verrà approfondito nelle sezioni 2.7.1, 2.7.2 e 2.7.5.

– `WEB-INF`. Cartella che contiene i file che servono al *servlet container*. Maggiori informazioni sono fornite nel capitolo 3..

Nella cartella principale esiste un file `build.xml` che è uno script ANT[4] per effettuare il *deploy* dell'applicazione.

Come prima azione di personalizzazione, va modificato il file `build.xml` inserendo al posto dei segnaposto `@NOME_PROGETTO@` e `@SERVER_HOME@` rispettivamente il nome che si vuole dare alla nuova applicazione Serena (nel nostro caso `pippo`) e il percorso per raggiungere la cartella principale del *servlet container*.

[3] Per approfondimenti si veda [Coo10].

[4] ANT è un linguaggio di scripting per la compilazione e/o il *deploy* di applicazioni. Per maggiori informazioni si veda [Hol05].

2.1.2 Modellazione dell'ontologia

Il passo fondamentale di personalizzazione dell'applicazione è la modellazione dei dati che l'applicazione deve gestire. Come visto nel capitolo 1, i dati gestiti dalle applicazioni Serena, pur essendo memorizzati fisicamente in un database SQL, vengono gestiti in tutto il loro ciclo di vita secondo un modello ontologico basato su *frame*.

Si intende per *frame* un oggetto primitivo che rappresenta un'entità nel dominio da rappresentare. Esistono frame chiamati *classi*, che rappresentano tipologie di entità (ad esempio la classe `Paziente` indica i tipi di dato che descrivono pazienti), e frame chiamati *istanze*, che rappresentano singoli "individui" (ad esempio `Mario Rossi` e `Agnese Corsaro` sono istanze della classe `Paziente`). Ogni classe può avere uno o più attributi chiamati *slot*.

Facendo un parallelismo con l'SQL, una classe può essere vista come una tabella, uno slot come un campo di una tabella e un'istanza come un record della tabella. Ogni slot può essere di tipo *ground*, cioè contenere valori foglia, oppure *relazionale*, cioé rappresentare una relazione tra la classe corrente e un'altra determinata classe (ad esempio una classe `Paziente` può avere uno slot relazionale `visite` che mette in relazione la classe `Paziente` con la classe `Visita`). Quanto detto fa saltare subito agli occhi il grande vantaggio di un'ontologia per la rappresentazione dei dati, rispetto al modello classico SQL: la possibilità di avere una struttura ad albero[5] (purché si ponga volta per volta una classe come radice). La figura 2.1 esemplifica quanto detto.

Chiarito questo aspetto, si può partire con la modellazione. Per far ciò si utilizza *Protégé*, un software sviluppato dall'Università di Stanford per la modellazione di ontologie. Si può vedere una schermata di Protégé in funzione nella figura 2.2.

L'utilizzo di Protégé è abbastanza intuitivo e una guida approfondita sull'uso di Protégé esula dagli scopi di questa tesi[6]. Qui basti sapere che

[5] La differenza tra modello ontologico e modello relazionale verrà approfondita nella sezione 5.1.

[6] Per approfondimenti si veda [oS10].

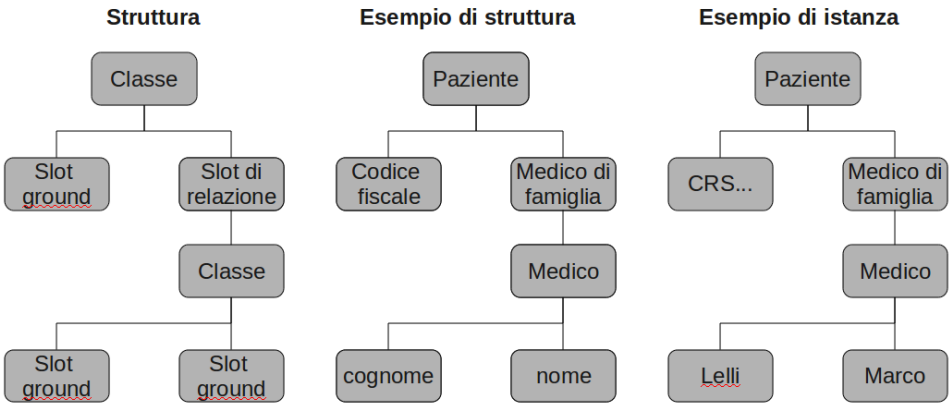

Figura 2.1: Esempio di ontologia

attraverso Protégé è possibile creare le classi e gli slot dell'ontologia dell'applicazione Serena[7], che non sono supportate da Serena le istanze e i vincoli creati direttamente da Protégé e che è necessario in Serena che ogni relazione tra una classe e un'altra sia indicata in entrambe le classi, ciascuna con il proprio slot di relazione[8].

Una volta modellata l'ontologia, è possibile generare i Serena Bean, i file di configurazione interpretabili da Serena, come mostrato nella sezione successiva.

2.1.3 Generazione dei Serena Bean

I Serena Entity Bean e i Serena Interface Bean sono i file di configurazione di base rispettivamente della struttura dei dati e della loro rappresentazione grafica. La loro configurazione dettagliata verrà mostrata rispettivamente nelle sezioni 2.6 e 2.7.6.

[7]Gli slot di Serena possono essere di una gamma più ampia di tipi rispetto agli slot di Protégé. L'argomento verrà trattato nella sezione seguente.

[8]Un approfondimento su come e quanto Serena supporta le ontologie create con Protégé è consultabile nella sezione 5.1.

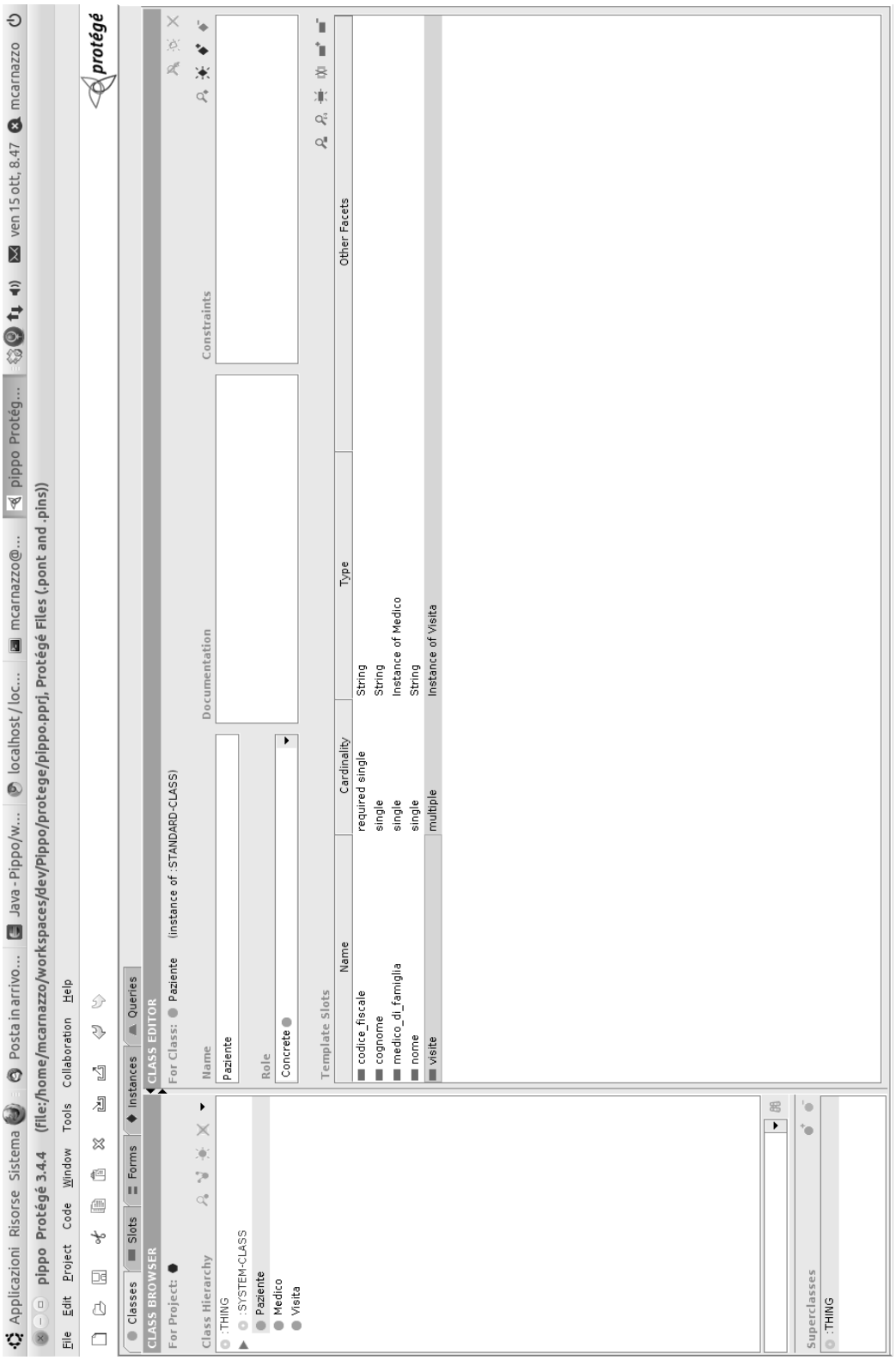

Figura 2.2: Schermata di Protégé in funzione

Per crearli si utilizza il *Serena Developer Tool*, un software sviluppato dalla Cooperativa Anastasis che attraverso un *wizard* trasforma le ontologie Protégé in Serena Bean.

Per avviarlo basta fare doppio click sul file `lib/sdt.jar` del nostro progetto Java in Eclipse.

Nella prima schermata (si veda la figura 2.3), dopo aver scelto il file Protégé da cui partire, il *wizard* mostra tutte le classi e gli slot dell'ontologia, dando la possibilità, per ogni slot di selezionarlo/deselezionarlo (per includerlo/escluderlo dall'applicazione) e di indicare il suo tipo di dato.

I tipi di dato Serena sono molti di più rispetto ai tipi di Protégé. La loro mappatura è la seguente:

- Gli slot `String` di Protégé possono diventare in Serena campi di tipo:
  - `Stringa` semplice
  - `Stringa criptata`, per gestire dati sensibili: tali stringhe vengono salvate su database previa criptazione[9]
  - `Password`, equivalenti alle stringhe criptate ma gestite applicativamente tramite campi password
  - `Numerico` (per ogni tipo di dato numerico: `int`, `float` ecc.)
  - `Data` per le date
  - `Data e ora` per i *timestamp*
  - `Check`, equivalenti ai *booleani*
  - `Importo`, per gestire le valute
  - `Email`, per gestire indirizzi email
  - `Http`, per gestire URL HTTP
  - `Immagine`, per gestire file immagini
  - `Audio`, per gestire file audio

[9]Per un approfondimento sulle stringhe criptate si veda 4.4.

  - `Video`, per gestire file video
  - `Attachment`, per gestire allegati generici
  - `Testo`, per gestire testi lunghi
  - `Editor`, per gestire testi lunghi con l'aiuto (a livello applicativo per gli utilizzatori delle applicazioni Serena) dell'editor HTML WYSIWYG PegoEditor[10]

- Gli slot `Symbol` di Protégé diventano in Serena le cosiddette *decodifiche*. Si tratta (in entrambi i sistemi) di campi il cui *range* di valori possibili è finito e definito.

- Gli slot `Instance` di Protegé sono slot relazionali: indicano una relazione tra una determinata classe e un'altra. In Serena vengono gestite tutte come relazioni bidirezionali: per ogni slot relazionale da una classe A a una classe B, deve esistere un rispettivo slot relazionale dalla classe B alla classe A. Le relazioni possono essere *dirette* o *inverse*. La differenza principale tra le due tipologie è che nei casi in cui in Serena i dati vengono gestiti in modo automatico (ad esempio quando vengono chiesti tutti i dati di una classe), il sistema non segue le relazioni inverse, al fine di non fare divergere gli algoritmi. In altre parole, gli algoritmi di visita di Serena, trattano le ontologie come grafi orientati. Un ulteriore suddivisione per le relazioni è in base alla molteplicità. Ci sono relazioni 1 a 1, 1 a N ed N ad M (a cui corrispondono come inverse rispettivamente relazioni 1 a 1, N a 1 ed N ad M).

Nella seconda schermata del *wizard* (si veda la figura 2.4) vengono gestiti gli elementi base della parte presentazionale: per ogni slot viene impostato il nome della relativa etichetta (che verrà utilizzata nelle pagine web dell'applicazione per indicare tale slot) e tutta una serie di informazioni aggiuntive

[10] Il PegoEditor è un editor HTML WYSIWYG orientato alla creazione di codice HTML accessibile e creato dalla Cooperativa Anastasis. Verrà descrito maggiormente nella sezione 4.3.

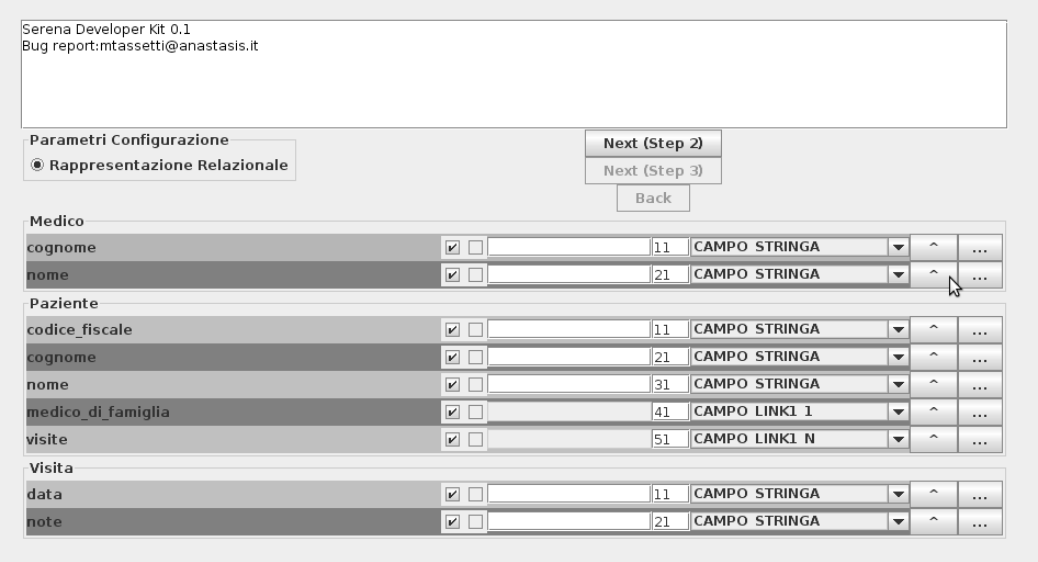

Figura 2.3: Prima schermata del Serena Developer Tool

(campi obbligatori nei moduli, campi considerati descrittivi dell'intera classe ecc.).

Nella terza e ultima schermata del *wizard* vengono visualizzati solo gli slot relazionali per stabilire quali relazioni sono dirette e quali inverse.

Terminato il *wizard*, il Serena Developer Tool genera o modifica i Serena Bean. Inoltre genera, all'interno della directory `offline_generated_files` del proprio progetto in Eclipse, alcuni file SQL per poter impostare/modificare il database su cui si appoggia l'applicazione Serena. Si può quindi passare alla configurazione del database e degli altri elementi di base dell'applicazione, come si vedrà nella sezione seguente.

2.1.4 Configurazione di base dell'applicazione

Terminata l'esecuzione del Serena Developer Tool, il progetto in Eclipse conterrà tutti i file necessari per un'applicazione Serena. Per poter avere un'applicazione Serena di base funzionante mancano ancora pochi passi, mostrati in questa sezione.

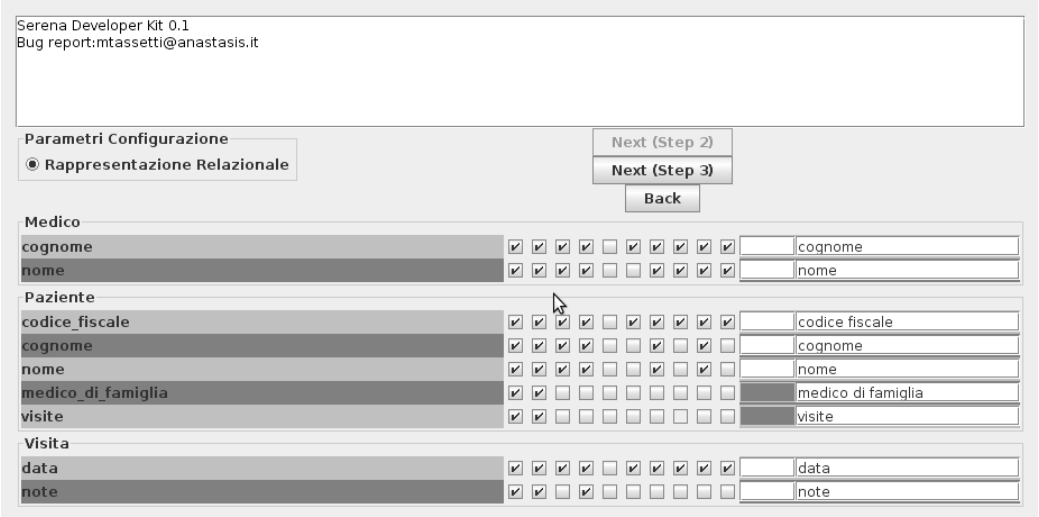

Figura 2.4: Seconda schermata del Serena Developer Tool

Bisogna innanzitutto creare un database SQL che conterrà i dati dell'applicazione. Le modalità di creazione di un database cambiano in base al *database server* utilizzato e per approfondire questo aspetto si invita a far riferimento a un manuale specifico del proprio *database server*.

Nel database creato bisogna importare alcuni file SQL generati dal Serena Developer Tool nella cartella `offline_generated_files`:

- `skeleton.sql`. Contiene i comandi per creare la struttura comune di tutte le applicazioni Serena.

- `<nomeprogetto>_db_create.sql` (es. `pippo_db_create.sql`). Contiene i comandi per creare, a partire dalla struttura comune, tutte le informazioni della nuova applicazione Serena.

- `<nomeprogetto_decodifiche_create.sql` (es. `pippo_decodifiche_create.sql`). Contiene i comandi per popolare il database con tutte le decodifiche ricavate dall'ontologia dell'applicazione Serena.

Da notare che nella stessa directory il Serena Developer Tool crea anche i file `<nomeprogetto>_db_update.sql` e `<nomeprogetto>_decodifiche_up-`

`date.sql`, per aggiornare rispettivamente la struttura del database e l'elenco delle decodifiche di un'applicazione Serena esistente.

Per concludere bisogna modificare tre file nella directory `app/conf/system`. Sono file in formato XML con una struttura autoesplicativa, corredati con ampia parte di commento che spiega ogni singolo parametro.

- `config_application.xml`. È il file di configurazione principale dell'applicazione Serena. Bisogna modificare il segnaposto `@NOME_APPLICAZIONE@` inserendo il nome dell'applicazione (es. `pippo`).

- `config_persistence.xml`. È il file di configurazione di Serena Persistence per l'interfacciamento con il database. Bisogna modificare almeno i segnaposto `@NOME_APPLICAZIONE@`, `@HOST_DB@`, `@NOME_DB@`, `@USER_DB@` e `@PASSWORD_DB@` inserendo rispettivamente il nome dell'applicazione e i dati di accesso al database (host, nome, username e password). Inoltre è possibile, seguendo le indicazioni nei commenti del file, indicare a quale tipo di *database server* bisogna interfacciarsi (la configurazione predefinita si interfaccia con database MySQL).

- `logservice.xml`. È il file di configurazione dei log, che segue lo standard di *Log4J*[11]. Bisogna modificare il segnaposto `@NOME_APPLICAZIONE@` inserendo il nome dell'applicazione.

Si conclude così la configurazione dell'applicazione Serena di base. Per poter visualizzare il risultato, bisogna effettuare il *deploy* nel *servlet container*. Per far ciò basta usare il *target* ANT `localDeploy` presente nel file `build.xml` nella directory principale del progetto in Eclipse. Una volta effettuato il *deploy* e avviato il *servlet container*, ammesso che sia in ascolto sulla porta 8080, l'applicazione Serena risponderà all'URL
`http://localhost:8080/<nomeprogetto>`
(ad esempio `http://localhost:8080/pippo`).

[11] Per approfondimenti si veda [Guc10].

Nella sezione successiva verrà mostrata l'applicazione Serena così creata, introducendo così alcune informazioni di base valide per ogni applicazione Serena.

2.2 L'applicazione Serena creata

Nella sezione precedente si è visto come creare un'applicazione di base chiamata Pippo. In questa sezione verrà mostrato il risultato ottenuto. Accedendo tramite browser all'URL `http://localhost:8080/pippo` verrà mostrata la pagina iniziale dell'applicazione, come mostrato nella figura 2.5.

L'interfaccia grafica è quella standard di un'applicazione Serena (si vedrà come personalizzarla nella sezione 2.7). Gli elementi mostrati, oltre a quelli grafici fissi di ogni pagina, sono il dettaglio di una pagina presentazionale, un *form* per effettuare ricerche all'interno del sito, un calendario di eventi e un *form* per effettuare il login. Si vedrà come personalizzare gli elementi non strettamente grafici nella sezione 2.5.

In modo predefinito esiste un utente `admin` (con password `admin`) per effettuare il login. Una volta effettuato il login viene mostrato anche un menu di navigazione con i comandi di amministrazione. Si vedranno nel dettaglio i comandi effettuabili dal menu di amministrazione nella sezione 2.3.

Supponendo di aver creato l'ontologia mostrata nella figura 2.1, dall'applicazione sarà possibile gestire oggetti di classe Paziente e di classe Medico. I dati che vengono mostrati nelle varie pagine dipendono da ciò che è stato impostato tramite il Serena Developer Tool descritto nella sezione 2.1.3.

I comportamenti possibili nelle applicazioni Serena e il modo di gestire gli oggetti verranno approfonditi nella sezione 2.5. Qui basti sapere che si possono gestire gli oggetti di una determinata classe partendo dal loro filtro di ricerca e che si accede al filtro di ricerca di una classe tramite l'URL `http://.../<nomeapplicazione>?q=object/filter&p=<nomeclasse>`
(ad esempio

Figura 2.5: Pagina iniziale dell'applicazione

`http://localhost:8080/pippo/?q=object/filter&p=Paziente`).
Verrà mostrata una pagina simile a quella della figura 2.6.

Dal filtro di ricerca è possibile effettuare ricerche sui dati della classe indicata oppure (se si hanno i permessi necessari) creare una nuova istanza della medesima classe. I risultati della ricerca vengono mostrati in una lista simile a quella mostrata in figura 2.7.

Dalla lista è possibile andare nel dettaglio di ogni istanza trovata oppure (se si hanno i permessi necessari) modificare uno degli oggetti trovati oppure crearne uno nuovo della medesima classe. Se si seleziona il dettaglio di un'istanza viene mostrata una scheda di dettaglio simile a quella mostrata in figura 2.8.

Dal dettaglio di un'istanza è possibile andare alla scheda di modifica della stessa istanza. Se si seleziona viene mostrata una scheda di modifica simile a quella mostrata in figura 2.9.

Dalla scheda di modifica è possibile modificare uno o più slot dell'istanza selezionata oppure cancellare l'intera istanza.

Così si possono già gestire tutti i dati dell'applicazione. Certo c'è ancora ampio margine di miglioramento, con molte personalizzazioni possibili, alcune direttamente dall'applicazione stessa (permessi e contenuti) altri tramite la modifica dei file di configurazione. Tutte le possibili personalizzazioni sono mostrate nelle sezioni successive.

2.3 Personalizzazione dei contenuti

Dall'applicazione è possibile effettuare una serie di personalizzazione sui contenuti dell'applicazione stessa attraverso il menu *Amministrazione*. Per la precisione è possibile gestire:

- *Pagine*: le pagine web di solo testo dell'applicazione, come ad esempio la pagina di presentazione iniziale.

Figura 2.6: Filtro di ricerca tipico dell'applicazione

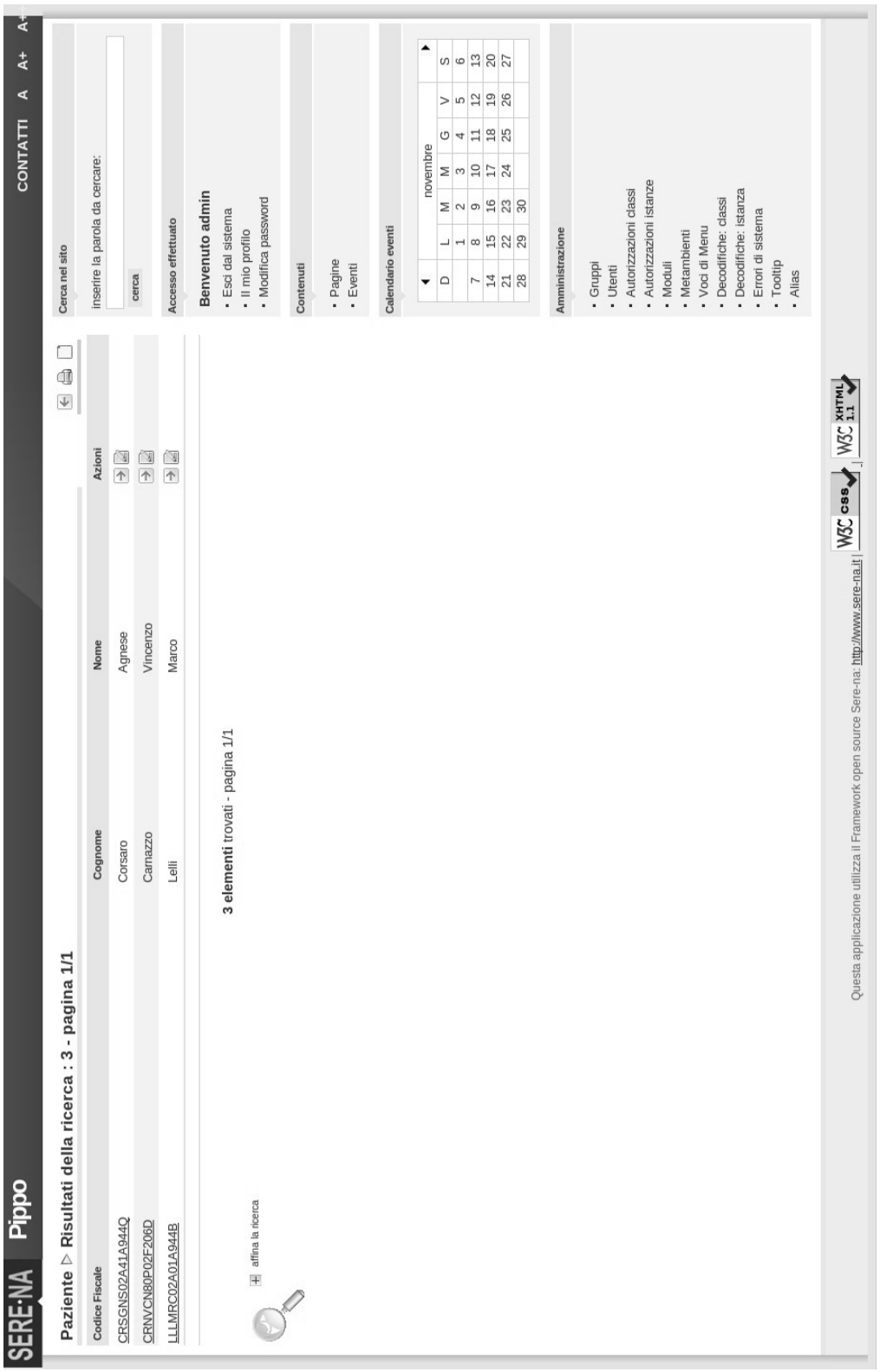

Figura 2.7: Lista di oggetti tipica dell'applicazione

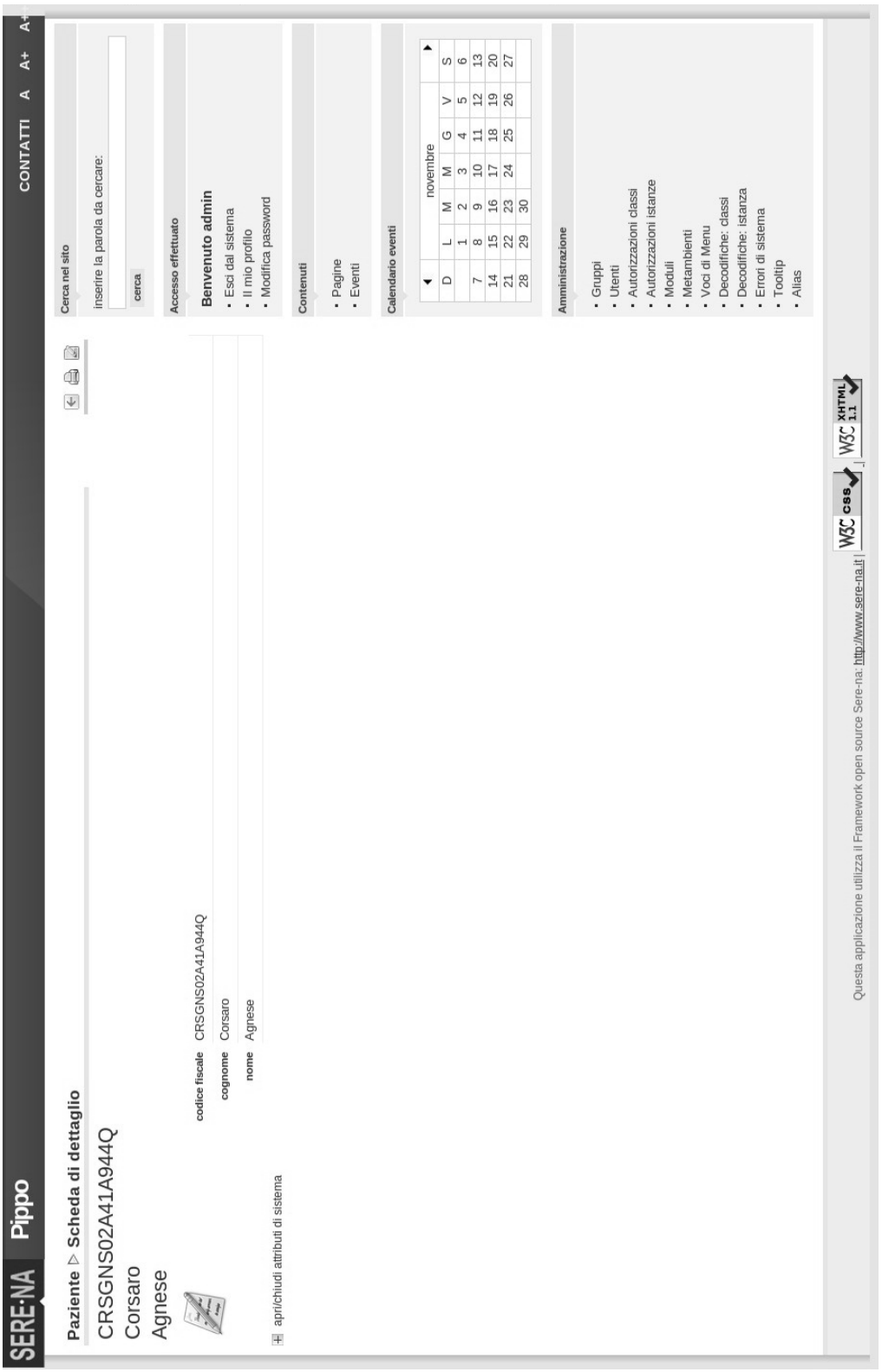

Figura 2.8: Scheda di dettaglio di un'istanza tipica dell'applicazione

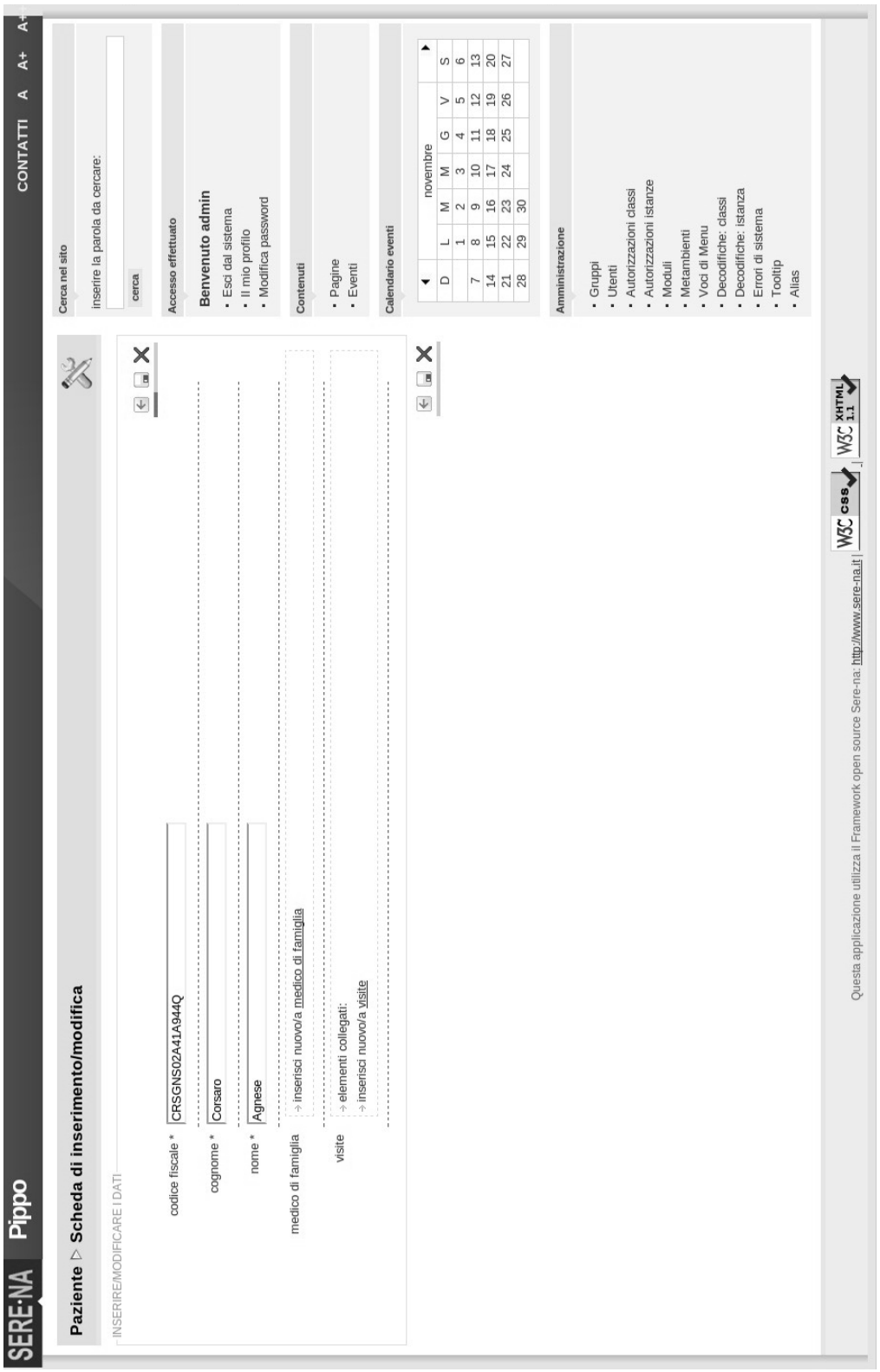

Figura 2.9: Scheda di modifica di un'istanza tipica dell'applicazione

- *Voci di menu*: le voci di menu presenti nell'applicazione. Quelle relative alle pagine del punto precedente possono essere gestire direttamente all'interno della gestione delle pagine.

- *Decodifiche*: le parti di testo presenti nei menu a tendina nei campi di decodifica, cioè nei campi che hanno valori stringa su un range ristretto di scelte.

- *Messaggi di errore*: le scritte da mostrare nelle varie situazioni di errore dell'applicazione.

- *Tooltip*: le scritte da mostrare negli help contestuali dell'applicazione.

Nella stessa voce di menu esistono anche *Utenti*, *Gruppi*, *Autorizzazioni di classe* e *Autorizzazioni di istanza*, che verranno descritte nel dettaglio nella sezione successiva, *Moduli*, che verrà descritta nel dettaglio nella sezione 2.5 e *Meta-ambienti*, che verrà descritta nel dettaglio nella sezione 2.7.

2.4 Personalizzazione dei permessi

Spesso un requisito fondamentale di un'applicazione web è la profilazione dei permessi di accesso e modifica dei contenuti. Per gestire queste situazione, è possibile all'interno di ogni applicazione Serena creare più utenti ed eventualmente raggrupparli in gruppi di utenti.

In modo predefinito in Serena esistono gli utenti seguenti:

- *everyone*. L'utente che naviga nell'applicazione senza effettuare il login.

- *admin*. L'unico utente (in prima battuta), che può effettuare il login, usando come password *admin*.

Inoltre in modo predefinito esistono i gruppi:

- *everyone*. Il gruppo dell'utente *everyone*, con permessi di lettura sui dati presentazionali e nessun permesso di scrittura.

- *admin*. Il gruppo dell'utente *admin*, con tutti i permessi.

- *operator*. Un gruppo (in prima battuta senza utenti) che ha permessi di lettura e scrittura su tutti i contenuti e nessun permesso per il lato amministrativo.

- *limited operator*. Un gruppo (in prima battuta senza utenti) che ha permessi di lettura e scrittura su tutti i contenuti creati da sé stesso e nessun permesso per il lato amministrativo.

È possibile creare, modificare e cancellare utenti e gruppi (nonché metterli tra loro in relazione) attraverso le apposite voci del menu *Amministrazione* dell'applicazione Serena. La loro gestione segue lo stesso flusso della gestione di un qualsiasi oggetto Serena, come visto già nella sezione 2.2.

Ogni utente può appartenere a più gruppi e godrà sempre dei permessi più ampi disponibili in ogni circostanza. I permessi vengono stabiliti a livello di gruppo.

È inoltre possibile stabilire permessi per singole classi, permessi che vanno sempre ad ampliare i permessi dei gruppi. È possibile infine stabilire permessi su singole istanze; in tal caso tali permessi avranno precedenza su qualsiasi altro permesso. Tali eccezioni vengono gestite tramite le voci di menu *Autorizzazioni su classi* e *Autorizzazioni su istanze* del menu *Amministrazione* dell'applicazione Serena. La loro gestione segue lo stesso flusso della gestione di un qualsiasi oggetto Serena, come visto già nella sezione 2.2.

Ogni autorizzazione è espressa attraverso una terna di valori, dove ogni valore è relativo rispettivamente al singolo utente, ai colleghi e a tutti[12].

Per colleghi di un utente si intende gli utenti che appartengono a uno o più gruppi a cui appartiene anche tale utente. Ciascun valore della terna può essere:

- `0`. Non ha permessi.

[12]Come si può facilmente notare, la gestione dei permessi delle applicazioni Serena imita quella dei sistemi operativi **nix like*.

- `1`. Permessi di sola lettura.
- `2`. Permessi di lettura e scrittura ma non di pubblicazione.
- `3`. Permessi di lettura, scrittura e pubblicazione.

Inoltre per ogni gruppo esiste un *flag* che indica se si ha il permesso di accedere a oggetti non pubblicati e di pubblicarli. Ciò permette un *workflow* di pubblicazione: alcuni utenti possono creare oggetti che però saranno visibili solo previa autorizzazione di altri utenti che ne hanno il diritto.

Per chiarire meglio, si mostrano qui di seguito alcuni esempi:

- `111`. L'utente può leggere tutto ma non può modificare niente.
- `333`. L'utente ha i permessi di un amministratore: può fare tutto.
- `222`. L'utente ha i permessi di un normale operatore: può leggere, creare e modificare tutto ma i suoi oggetti creati saranno resi pubblici solo se autorizzati da altri.
- `110`. L'utente può leggere solo gli oggetti creati da sé stesso o da utenti dei suoi stessi gruppi. Non può modificare niente.
- `210`. L'utente può leggere gli oggetti creati da sé stesso e può leggere quelli dei suoi colleghi. Può creare oggetti e può modificare solo quelli creati da sé stesso. Non ha alcun permesso sugli oggetti creati da utenti di gruppi a cui non appartiene.
- `300`. L'utente ha tutti i permessi ma solo relativamente ai propri oggetti.
- `321`. L'utente ha tutti i permessi sui propri oggetti, permessi di lettura e modifica sugli oggetti dei suoi colleghi e solo permessi di lettura sugli oggetti creati da utenti di altri gruppi.

A ulteriore chiarimento si pensi allo scenario reale di un'applicazione web per il personale amministrativo di un ospedale. Gli utenti del gruppo *limited operator* potranno vedere tutti i dati ma non potranno modificare nulla tranne i pazienti creati da loro stessi. In questo scenario il gruppo *limited operator* avrà permessi `111` e sulla classe `Paziente` ci saranno permessi `211`.

Stabilito come personalizzare i permessi, si vedrà nella prossima sezione come personalizzare i comportamenti dell'applicazione, cioé cosa concretamente l'applicazione "sa fare".

2.5 Personalizzazione dei comportamenti: i Serena Module

I comportamenti possibili all'interno di un'applicazione Serena sono stabiliti dai cosiddetti *Serena Module*. I Serena Module sono dei plugin installabili in ogni applicazione Serena e richiamabili esplicitamente dall'utente passando gli opportuni parametri tramite l'URL del browser oppure automaticamente dall'applicazione attraverso opportuna configurazione del sistema.

Ogni modulo può occuparsi di diverse funzioni e fornice quindi più metodi. Ad esempio, il modulo `object` (il modulo principale di Serena) si occupa della manipolazione degli oggetti ed ha tra i suoi metodi il metodo `filter` per creare un filtro di ricerca.

La chiamata esplicita avviene aggiungendo all'URL dell'applicazione il seguente parametro [13]:

?q=<nomemodulo>/<nomemetodo>

[13] Purtroppo la sintassi delle chiamate non rispettano lo standard URI, in quanto i caratteri `=` e `/` sono riservati: per utilizzarli andrebbe prima effettuato un *URL encoding* (andrebbero cioè sostituiti rispettivamente con `%3D` e `%2F`). La chiamata corretta sarebbe dunque `?q=<nomemodulo>%2F<nomemetodo>`. Fortunatamente i browser moderni effettuano l'*URL encoding* automaticamente (tant'è che lo stesso aspetto negativo è comune ad altre applicazioni web, come ad esempio Drupal).

Qualora si vogliano chiamare più moduli contemporaneamente, basta concatenare le chiamate separandole con `&`. Esempio:

```
?q=<nomemodulo1>/<nomemetodo1>
&q=<nomemodulo2>/<nomemetodo2>
```

Ad ogni modulo è possibile passare dei parametri. Essendo ogni modulo un plugin che fa storia a sé, possono esistere vari modi di passaggio dei parametri, anche se di fatto le sintassi sono due:

```
?q=<nomemodulo1>/<nomemetodo1>
&q=<nomemodulo2>/<nomemetodo2>
&<param1>=<valore1>
&<param2>=<valore2>
&<param3>=<valore3>
```

oppure:

```
?q=<nomemodulo1>/<nomemetodo1>
/<param11>=<valore11>
/<param12>=<valore12>
/...
?q=<nomemodulo2>/<nomemetodo2>
/<param21>=<valore21>
/<param22>=<valore22>
/...
```

La prima modalità è utile se i parametri vanno condivisi tra più moduli, mentre la seconda se devono essere specifici per un modulo. Il modulo `object` usa la prima sintassi[14].

Ad esempio, per richiedere un filtro di ricerca per i dati della classe `Paziente` bisogna usare i seguenti parametri:

[14] In realtà la prima modalità è ormai deprecata e viene mantenuta solo per retrocompatibilità. Non è possibile mescolare le due modalità in quanto i singoli moduli si aspettano i parametri in una o nell'altra modalità. Per ovviare a questo problema è già in fase di sviluppo, per la futura versione di Serena, la possibilità di passare i parametri indifferentemente nella prima o nella seconda modalità.

?q=object/filter&p=Paziente

La chiamata automatica avviene tramite modifica della configurazione del modulo, attraverso l'apposita voce del menu *Amministrazione* dell'applicazione Serena. La sua gestione segue lo stesso flusso della gestione di un qualsiasi oggetto Serena, come visto già nella sezione 2.2.

Ogni applicazione Serena nativamente ha alcuni moduli installati. Le modalità d'uso dei singoli moduli e dei relativi metodi va oltre gli scopi di questa tesi[15]. Qui a scopo esemplificativo viene mostrata la lista dei moduli nativi e dei relativi metodi principali:

- Modulo `object`. È il modulo principale di Serena. Si occupa di tutta la manipolazione dei dati. È stato sviluppato principalmente da Andrea Frascari, con molteplici interventi di Vincenzo Carnazzo, Matteo Tassetti e Andrea Pegoretti. I suoi metodi principali sono:
  - `filter`. Mostra una *form* per la ricerca dei dati. La classe su cui lavorare è stabilita dai parametri passati al metodo. Esempio di utilizzo per richiedere un filtro di ricerca per i dati della classe `Paziente`:

    ?q=object/filter&p=Paziente

  - `list`. Mostra una lista di istanze. Solitamente è il risultato ottenuto inviando la *form* creata con il metodo `filter`. Esempio di utilizzo per mostrare tutti i dati della classe `Paziente`:

    ?q=object/list&p=Paziente

  - `detail`. Mostra la scheda dettagliata di un'istanza. La classe su cui lavorare e i criteri per identificare l'istanza da mostrare sono stabiliti dai parametri passati al metodo. Esempio di utilizzo per mostrare la scheda del paziente con `ID=1`:

    ?q=object/detail&p=Paziente/_a_ID/_v_1

[15] Per approfondimenti si veda [Coo10].

  - `detail_edit`. Se chiamato con una chiamata HTTP GET mostra una *form* per la modifica/cancellazione di un'istanza. La classe su cui lavorare e i criteri per identificare l'istanza da mostrare sono stabiliti dai parametri passati al metodo. Se chiamato con una chiamata HTTP POST effettua le modifiche richieste. Le modifiche effettuate sono stabilite dai parametri passati al metodo. Esempio di utilizzo per mostrare la modifica dei dati del paziente con `ID=1`:

    ```
    ?q=object/detail_edit&p=Paziente/_a_ID/_v_1
    ```

- Modulo `expertsystem`. Si occupa di integrare i dati dell'ontologia con il Sistema Esperto Drools. Concretamente gestisce sessioni di Drools dedicate a una determinata classe e le sue relative relazioni, sulla quale è possibile richiedere inferenze in base a regole prestabilite[16]. I suoi metodi sono:

  - `startSession`. Inizializza una sessione di Drools dedicata a un oggetto principale (e relative relazioni, se necessario)
  - `closeSession`. Chiude la sessione dedicata a un oggetto, cancellando i suoi dati in memoria.
  - `viewExpertSystem`. Mostra le inferenze prodotte dal motore inferenziale per uno specifico oggetto della classe principale.
  - `viewInference`. Mostra il dettaglio di una singola inferenza.

- Modulo `login`. Si occupa dell'accreditamento degli utenti registrati. È stato sviluppato principalmente da Matteo Tassetti. Ha due metodi:

  - `login`. Accetta come parametri username e password e si occupa di controllare che un utente con tali credenziali esista. Se così è,

[16] Lo sviluppo del modulo `expertsystem` è frutto di una tesi di laurea della Facoltà di Ingegneria dell'Università di Bologna. Per approfondimenti si veda [Azz08]. Inoltre sull'uso di Drools si veda [Bal09].

registra l'utente nella sessione HTTP corrente e lo fa accedere al sistema, altrimenti mostra un messaggio di errore.

  - `logout`. Cancella l'utente corrente dalla sessione e lo porta alla pagina iniziale dell'applicazione. Non ha parametri.

- Modulo `menu`. Mostra un frammento di HTML con all'interno un menu di navigazione. È stato sviluppato principalmente da Matteo Tassetti. Le varie voci di menu sono contenute all'interno della classe di sistema `_system_menu_item`, la cui gestione è delegata al modulo Object. Il modulo Menu ha un solo metodo, chiamato anch'esso `menu`, che prende come parametro l'ID dell'istanza di `_system_menu_item` considerata padre di tutto il menu di navigazione.

- `report`. Generare reportistica in vari formati a partire dai dati dell'applicazione. È stato sviluppato principalmente da Vincenzo Carnazzo. Al momento i formati supportati sono PDF, DOC, XLS, PPT e HTML. Ha due metodi:

  - `call`. Mostra un frammento di HTML con i link per generare la reportistica.

  - give. Genera un report. La classe su cui lavorare, il formato da usare e i criteri per identificare le istanze da mostrare sono stabiliti dai parametri passati al metodo. La generazione dei report avviene a partire da *report design* creati con l'IDE Birt, a cui il sistema in fase di chiamata del metodo *give* passa l'XSerena da usare come *data source XML*[17]. Esempio di utilizzo per creare un report PDF chiamato relazione relativamente al paziente con `ID=1`:

    ?q=reporg/give/CLS=Paziente/ID=1
    /DOC=relazione/TYPE=pdf

[17]Per approfondire il funzionamento e l'utilizzo di Birt si veda [HPT08].

- Modulo `stat`. Genera grafici statistici sui dati presenti nell'applicazione. È stato sviluppato principalmente da Matteo Tassetti. Ha due metodi:

  - `distribution`. Mostra un grafico di distribuzione. La classe su cui lavorare e i criteri per identificare le istanze da mostrare sono stabiliti dai parametri passati al metodo. Esempio di utilizzo per avviare il *wizard* per mostrare un grafico sui dati di classe `Paziente`:

    ?q=stat/distribuzion/CLS=Paziente

  - `timeseries`. Mostra un grafico temporale. La classe su cui lavorare e i criteri per identificare le istanze da mostrare sono stabiliti dai parametri passati al metodo. Esempio di utilizzo per avviare il *wizard* per mostrare un grafico sui dati di classe `Paziente`:

    ?q=stat/timeseries/CLS=Paziente

Si noti inoltre che Serena è sviluppata in modo modulare e che ogni sviluppatore può creare propri Serena Module. L'argomento verrà ripreso e approfondito nella sezione 3.5.

2.6 Personalizzazione della base di dati

Si è visto nella sezione 2.1.3 come configurare la struttura dei dati con il Serena Developer Tool. In realtà il Serena Developer Tool è solo un tool che facilita la creazione dei file di configurazione veri e propri, che nel caso della struttura dei dati sono i Serena Entity Bean, che verranno illustrati in questa sezione.

I Serena Entity Bean si trovano all'interno della directory `webapps/app/conf/entities`. Ogni classe dell'ontologia ha il suo relativo Serena Entity Bean, il cui nome è `<nomeclasse>.xml`. La struttura generica di ogni bean è la seguente:

```
<?xml version="1.0" encoding="UTF-8"?>

<bean>
  <title>@nomeclasse@</title>
  <data_source>@nometabelladb@</data_source>
  <order_by />
  <xml>
    <class>@nomeclasse@</class>
  </xml>
  <attributes>
    <item>
      <name>ID</name>
      <type>2</type>
      <xml>
        <name>ID</name>
      </xml>
    </item>
    <item>
      <name>@nomecampodb@</name>
      <type>@tiposlot@</type>
      <xml>
        <name>@nomeslot@</name>
        <class/>
      </xml>
    </item>
    <!-- .. -->
  </attributes>
</bean>
```

Dove:

- Al posto di `@nomeclasse@` va inserito il nome che la classe ha nell'ontologia.

- Al posto di `@nometabelladb@` va inserito il nome della tabella che nel database contiene le informazioni della classe.

- All'interno del tag `attributes` vanno inserite le informazioni di ogni slot, ciascuno all'interno di un tag `item`.

- Al posto di `@nomeslot@` va inserito il nome che lo slot ha nell'ontologia.

- Al posto di `@tiposlot@` va inserito il tipo di valore contenuto nello slot. Il tipo di valore è rappresentato attraverso un valore numerico: 1 sta per stringa, 2 sta per numero, 3 sta per data ecc.[18].

- Al posto di `@nomecampodb@` va inserito il nome del campo nel database che contiene le informazioni della classe.

Da notare che ogni Serena Entity Bean contiene sempre (anche se non presente nell'ontologia) uno slot *ID*, gestito internamente dall'applicazione, in modo trasparente all'utente, e che viene utilizzato ovunque come *foreign key*.

Alcuni slot più complessi (come le decodifiche o gli slot di relazione) avranno una sintassi leggermente più ampia e non verrà qui analizzata[19]. La modifica diretta dei Serena Entity Bean è comunque quasi sempre evitabile in quanto completamente gestibile attraverso il Serena Developer Tool.

Conclusa la personalizzazione di comportamenti e di dati da gestire, si può passare alla personalizzazione della grafica, analizzata approfonditamente nella sezione successiva.

2.7 Personalizzazione della grafica

In questa sezione verrà analizzato nel dettaglio come, a partire da dati grezzi, Serena prepara le pagine web che vengono mostrate all'utente. Se si

[18]Per l'elenco completo dei tipi di dati si veda [Coo10].

[19]Per approfondimenti si veda [Coo10].

ha chiaro questo aspetto, è possibile intervenire per personalizzare il risultato finale.

Ogni schermata di un'applicazione Serena è divisa in più elementi grafici, ciascuno approfondito qui di seguito. La suddivisione è mostrata nella figura 2.11.

Nella pagina sono evidenziati:

- Il *meta-ambiente*. È la parte grafica fissa della pagina (nella figura è la parte evidenziata in rosso); quella che solitamente contiene l'intestazione, il *footer* e in generale la struttura principale della pagina.

- Una parte relativa al contenuto di un singolo metodo di un modulo (nella figura la parte relativa al modulo `object` è evidenziata in giallo[20]). La grafica di queste parti è impostata nei cosiddetti *Serena Template*. Si tratta di scheletri grafici simili al JSP[21] con frammenti di HTML, parti di un mini-linguaggio interno di Serena e riferimenti ai dati da mostrare.

- Una parte relativa alla rapparesentazione grafica di un singolo dato (nella figura è la parte evidenziata in verde[22]). La grafica di queste parti è impostata nei cosiddetti *Serena Component*. Si tratta di scheletri grafici semplici con frammenti di HTML e riferimenti ad alcune impostazioni relative al singolo dato stabilite nei Serena Interface Bean (descritti nella sezione 2.7.6).

Vedremo tutti questi elementi nel dettaglio qui di seguito.

[20] Per evitare confusioni, nella figura le parti relative ad altri moduli sono stati offuscati.

[21] JSP sta per *Java Server Pages* ed è una tecnologia Java per fornire contenuti dinamici in formato HTML. Per maggiori informazioni si veda [Goo00].

[22] Nella pagina sono presenti naturalmente altri dati. Nella figura ne è stato evidenziato solo uno a titolo esemplificativo.

Figura 2.10: Una pagina di un'applicazione Serena

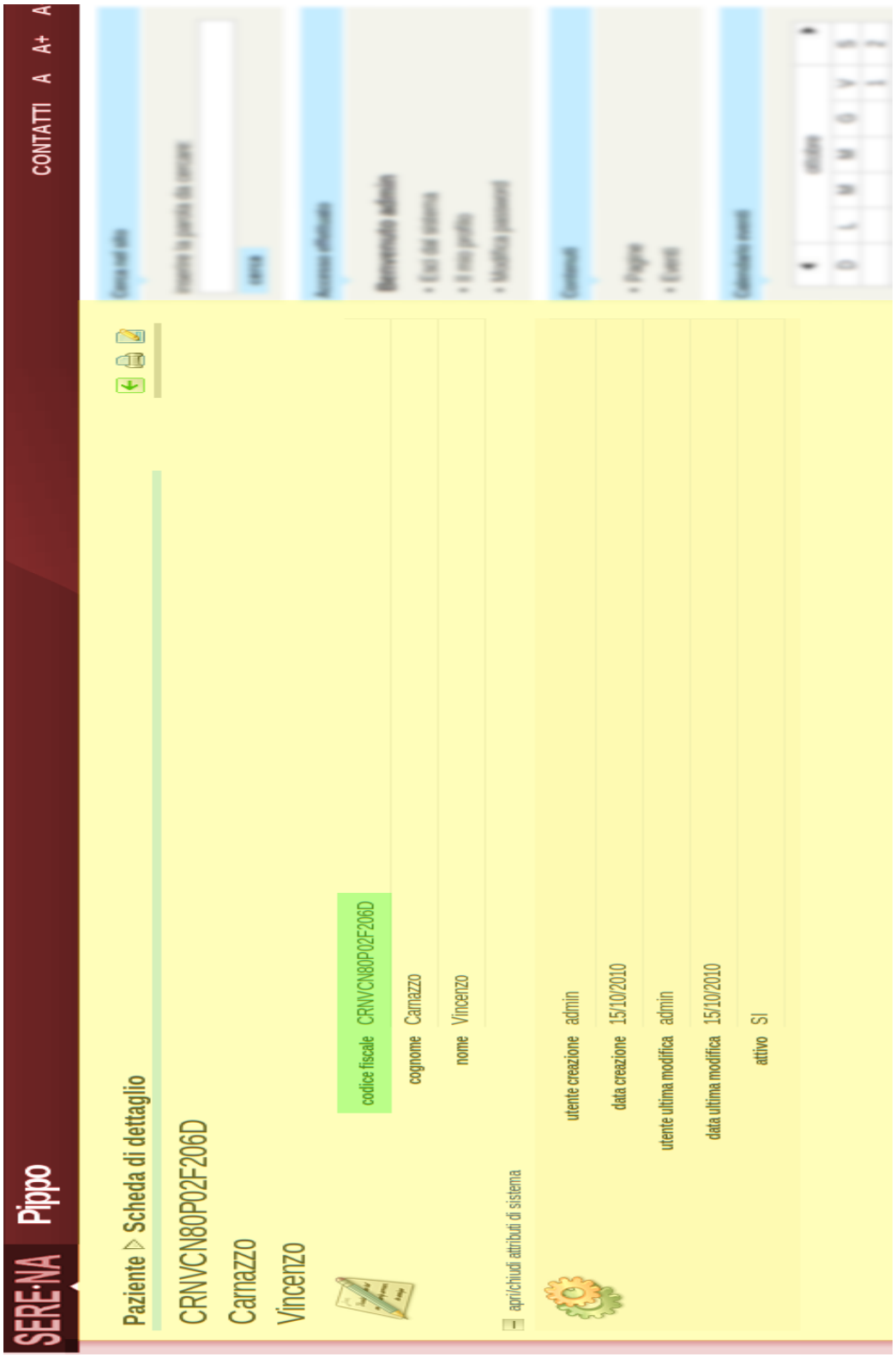

Figura 2.11: Una pagina di un'applicazione Serena suddivisa in parti

2.7.1 Meta-ambienti

Un meta-ambiente è lo scheletro grafico principale di un'applicazione Serena. Contiene il codice HTML principale e una serie di segnaposto che indicano dove devono posizionarsi i Serena Module invocati.

In ogni applicazione Serena possono esistere più meta-ambienti. Gli unici vincoli sono che:

- Esista almeno un meta-ambiente chiamato `standard`.
- Ogni meta-ambiente abbia le sue informazioni all'interno della directory `webapps/app/Template/metaAmbiente/<nomemetaambiente>`.
- Nella propria directory esista almeno un file chiamato `theme.htm` contenente appunto lo scheletro.

Un frammento esemplificativo di file `theme.htm` di un meta-ambiente è il seguente:

```
<html>
  <head>
    <title>NOME APPLICAZIONE</title>
    <!-- ... -->
  </head>
  <body id="home" class="normale">
    <!-- ... -->
    <div id="nav">
      <!-- NAVIGATION MENU -->
      @MOD_0@
    </div>
    <!-- ... -->
    @MOD_1@
    <div id="sidebar">
      @MOD_2@
      <!-- SEARCH BOX -->
```

```
        @MOD_3@
        <!-- MENU BOX -->
        @MOD_4@
        <!-- NEWS BOX -->
      </div>
      <div id="content">
        <!-- MAIN CONTENT -->
        @MOD_5@
      </div>
      <div id="footer" class="clearfix">
        @MOD_6@
        <p id="credits">
          <!-- ... -->
        </p>
      </div>
    </body>
</html>
```

Come si può notare, è un normale file HTML con l'unica particolarità di aver in alcuni punti alcuni tag nella forma `@MOD_x@`: sono dei segnaposto dove si collocherà il contenuto dei Serena Module. Ogni Serena Module, infatti, ha tra le informazioni della sua configurazione la *posizione* e l'*ordine*. Il contenuto di un Serena Module configurato per stare in posizione 1, verrà inserito al posto del tag `@MOD_1@`. Se più Serena Module sono configurati per stare nella stessa posizione, il loro ordine verrà stabilito in base al campo *ordine*. Da notare che ogni meta-ambiente può avere moduli diversi, in posizioni e ordine diversi.

È possibile creare, modificare e cancellare meta-ambienti attraverso l'apposita voce del menu *Amministrazione* dell'applicazione Serena. La sua gestione segue lo stesso flusso della gestione di un qualsiasi oggetto Serena, come visto già nella sezione 2.2.

Solitamente il sistema usa il Meta-ambiente `standard`. Il cambio di Meta-

ambiente avviene tramite richiesta esplicita da URL, attraverso il parametro `me`, come da esempio:

?me=<nomealtrometaambiente>

2.7.2 Serena Template

Il risultato dell'invocazione di un Serena Module è solitamente un frammento di codice HTML da mostrare in un determinato punto del meta-ambiente, come visto nella sezione precedente. Lo scheletro di questo frammento di codice è stabilito dai cosidetti Serena Template, descritti qui di seguito.

Un Serena Template è sempre relativo al metodo di un Serena Module applicato a una specifica classe e viene memorizzato all'interno del file `app/Template/<nomemodulo>/<nomemetodo>/<nomeclasse>.htm` dell'applicazione Serena. Ad esempio, il Serena Template visto come esempio precedentemente per la scheda di dettaglio di un paziente, sarà memorizzato nel file `app/Template/object/detail/Paziente.htm`.

Un esempio di Serena Template per il dettaglio di un'istanza è il seguente:

```
@BEGIN_TEMPLATE@
  <div class="title">
    <h3>Scheda di dettaglio</h3>
  </div>
  <div class="content">
    @BEGIN_Paziente@
      @tag_codice_fiscale@

      <hr />

      <h4>Medico di famiglia:</h4>
      @BEGIN_medico_di_famiglia@
        @BEGIN_Medico@
```

```
            @tag_cognome@
            @tag_nome@
          @END_Medico@
        @END_medico_di_famiglia@

        <hr />
        <p>La prima visita &egrave; stata effettuata il
            @tag_prima_visita.Visita.data@</p>
      @END_Paziente@
    </div>
@END_TEMPLATE@
```

Le caratteristiche salienti di un Serena Template sono:

- Deve iniziare con `@BEGIN_TEMPLATE@` e terminare con `@END_TEMPLATE@`.
- Può contenere al suo interno codice HTML sparso. Non contiene la struttura principale di un documento HTML (come ad esempio i tag `html` o `body` in quanto lavora esclusivamente su frammenti di codice: le parti principali del documento HTML sono all'interno del meta-ambiente, visto nella sezione 2.7.1.
- Al suo interno è possibile navigare l'ontologia dei dati, nel modo seguente:
  - Il frammento relativo a una classe va racchiuso all'interno dei tag `@BEGIN_nomeclasse@` ed `@END_nomeclasse@`.
  - Il frammento relativo a un'istanza di relazione va racchiuso all'interno dei tag `@BEGIN_nomeslot@` ed `@END_nomeslot@`.
  - Il frammento relativo a un valore ground va richiesto scrivendo `@tag_nomeslot@`.
  - Il frammento relativo a un valore ground di uno slot raggiungibile solo all'interno di uno slot di relazione va richiesto scrivendo `@tag_percorso.per.arrivare.all.istanza.nomeslot@`.

Per "navigare nell'ontologia" si intende che a ogni segnaposto relativo a un dato viene sostituito il dato stesso (con eventualmente qualche aggiunta grafica, come si vedrà nella sezione 2.7.5). Nel caso vengano gestiti più dati, ogni "navigazione nell'ontologia" equivale a un ciclo *for* sui dati.

Ad esempio, il metodo `list` del modulo `object` (che mostra una lista di oggetti) avrà un Serena Template simile al seguente:

```
@BEGIN_TEMPLATE@
  <div class="title">
    <h3>Lista</h3>
  </div>
  <div class="content">
    <table>
      @BEGIN_Paziente@
        <td>@tag_codice_fiscale@</td>
        <td>@tag_cognome@</td>
        <td>@tag_nome@</td>
      @END_Paziente@
    </table>
  </div>
@END_TEMPLATE@
```

All'interno dei Serena Template è possibile inoltre utilizzare lo standard XPath 1.0[23]. Il suo utilizzo può avvenire in due modi:

- È possibile utilizzarlo per scremare le istanze da mostrare. Per farlo, si aggiunge la condizione di filtraggio, seguendo lo standard XPath, all'interno del tag `@BEGIN_xxx@`. Ecco ad esempio come mostrare soltanto i pazienti il cui cognome è Rossi:

[23] XPath è uno standard del W3C per la creazione di percorsi all'interno di un documento XML. Non è previsto nell'immediato il supporto a XPath 2.0, in quanto le specifiche dello standard sono mutate notevolmente nel cambio di versione. Per maggiori informazioni si veda [Sim02].

```
@BEGIN_TEMPLATE@
  <!-- ... -->
      @BEGIN_Paziente[cognome='Rossi']@
        <td>@tag_codice_fiscale@</td>
        <td>@tag_cognome@</td>
        <td>@tag_nome@</td>
      @END_Paziente@
  <!-- ... -->
@END_TEMPLATE@
```

- È possibile utilizzare XPath che valutano alcuni elementi e ritornano una stringa. Per farlo, si usa la *XPath Function*. La XPath Function viene chiamata seguendo il formato `@XPATH_FUN(<xpath>)@`, dove `<xpath>` indica l'xpath da applicare. L'XPath viene applicato a partire dalla posizione che si ha nell'ontologia nel punto in cui la XPath Function viene chiamata. Se ad esempio si vuole calcolare il numero di visite effettuate da un paziente, si può usare il Serena Template seguente:

```
@BEGIN_TEMPLATE@
  <!-- ... -->
  @BEGIN_Paziente@
    Numero di visite:
    @XPATH_FUN(count(visite/Visita))@
  @END_Paziente
@END_TEMPLATE@
```

Spesso i Serena Template per un metodo sono molto simili per tutte le classi. In tal caso è possibile far uso dei cosiddetti *Serena Meta-Template*, ovvero dei template generici per creare template specifici.

Qui di seguito viene mostrato un esempio di Serena Meta Template per tutte le schede di dettaglio:

```
@METABEGIN_TEMPLATE@
```

```
@BEGIN_TEMPLATE@
  <div class="title">
    <h3>Scheda di dettaglio</h3>
  </div>
  <div class="content">
    @METABEGIN_bean@
      @BEGIN_@METAtag_title@@
        @METABEGIN_attributes@
          @METABEGIN_item[detail]@
            @tag_@METAtag_name@@
          @METAEND_item@
        @METAEND_attributes@
      @END_@METAtag_title@@
  </div>
@END_TEMPLATE@
@METAEND_TEMPLATE@
```

Un Serena Meta Template è molto simile a un Serena Template, con le seguenti particolarità:

- Deve iniziare con `@METABEGIN_TEMPLATE@` e terminare con `@METAEND_TEMPLATE@`.
- Al suo interno invece di navigare all'interno di un'ontologia, naviga all'interno del Serena Interface Bean[24] relativa alla classe richiesta.
- Nella navigazione, al posto dei tag `@BEGIN_...` ed `@END_...` si usano rispettivamente i tag `@METABEGIN_...@` e `@METAEND_...@`.
- Analogamente, per far riferimento a un tag del Serena Interface Bean, al posto di `@tag_...@` si usa `@tag_@METAtag_...@@`

[24] I Serena Interface Bean descrivono gli elementi grafici di base di ogni classe e verranno analizzati nel dettaglio nella sezione 2.7.6.

Applicato alla classe Paziente, il Meta Template di esempio indicato sopra (supponendo che nel Serena Interface Bean di Paziente siano indicati da inserire in `detail` solo gli slot `nome` e `cognome`) genererà il seguente Serena Template:

```
@BEGIN_TEMPLATE@
  <div class="title">
    <h3>Scheda di dettaglio</h3>
  </div>
  <div class="content">
    @BEGIN_Paziente@
      @tag_cognome@
      @tag_nome@
    @END_Paziente@
  </div>
@END_TEMPLATE@
```

Un Serena Meta Template va memorizzato all'interno del file `app/Template/metatemplate/<nomemodulo>/<nomemetodo>.htm` dell'applicazione Serena. Ad esempio, il Serena Meta Template per il dettaglio degli oggetti sarà memorizzato nel file `app/Template/metatemplate/object/detail.htm`.

Quando il sistema ha bisogno del template per un metodo applicato a una determinata classe, esso cercherà automaticamente il Serena Template specifico in `app/Template/<nomemodulo>/<nomemetodo>/<nomeclasse>.htm`; qualora non lo trovi lo creerà in automatico a partire dal Serena Meta Template in `app/Template/metatemplate/<nomemodulo>/<nomemetodo>.htm`.

Si può notare una certa analogia tra i Serena (Meta) Template e i fogli XSLT. I due sistemi verranno confrontati nella sezione 5.5.

Salvo per alcune classi di sistema, l'applicazione di base creata seguendo quanto descritto nella sezione 2.1 usa soltanto Meta Template. Per personalizzare l'applicazione spesso è necessario creare una serie di Serena Template specifici. Per creare tali Template facilmente, partendo da un Meta Template, è possibile usare il *Template Generator*, descritto nella sezione seguente.

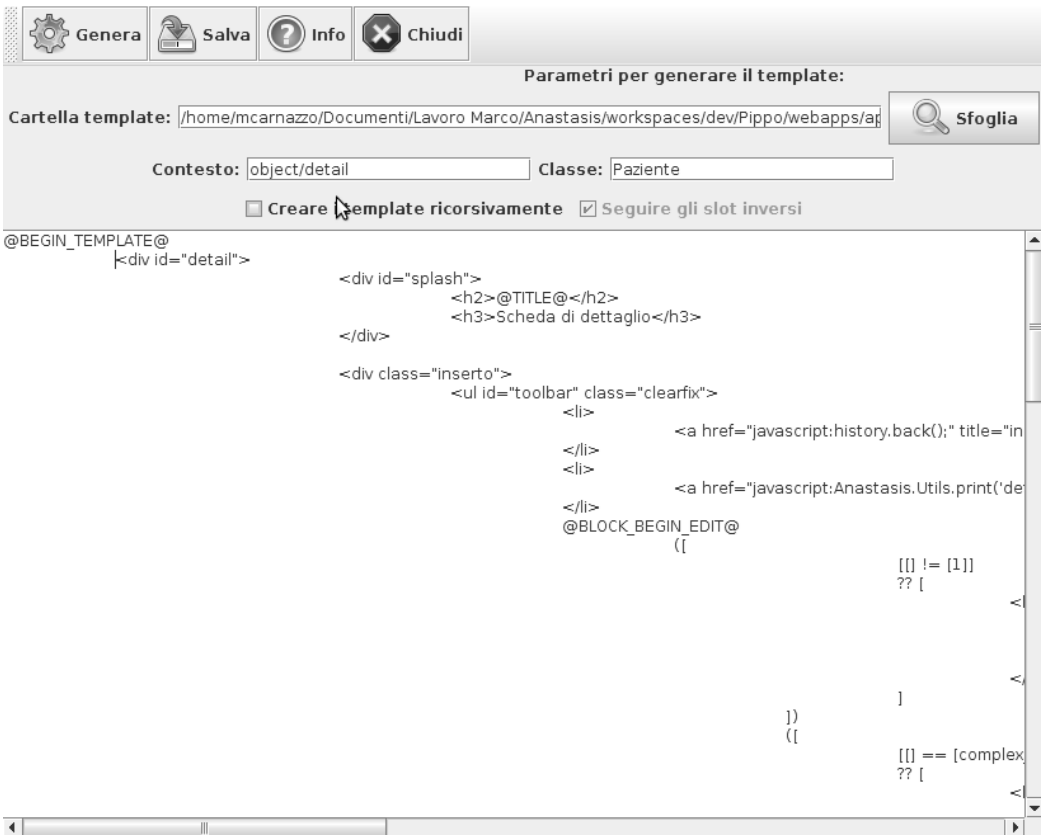

Figura 2.12: Schermata di esempio del Template Generator

2.7.3 Template Generator

Il Template Generator è un tool del Framework Serena per la generazione di Serena Template a partire da Meta Template. Per avviarlo basta fare doppio click sul file `lib/templategenerator.jar` del nostro progetto Java in Eclipse. La schermata del Template Generator è mostrata in figura 2.12.

Nel Template Generator basta indicare l'applicazione su cui si sta lavorando, il modulo, il metodo e la classe per i quali si vuole creare il Serena Template ed esso, seguendo la stessa logica di Serena, genera il codice del Serena Template, mostrandolo nell'anteprima al centro del programma. È quindi possibile salvare il Serena Template: il tool identificherà automaticamente il percorso dove salvare il template.

2.7.4 Programmazione avanzata nei Serena Template

Oltre a quanto già visto, nei Serena Template è possibile inserire alcuni costrutti di un mini-linguaggio creato ad hoc, nonché chiamate alle cosiddette Serena Function.

Questi elementi non verranno qui approfonditi[25]. A titolo esemplificativo vengono mostrati solo due esempi di utilizzo.

Un esempio di utilizzo del mini-linguaggio dei Serena Template è il seguente[26]:

```
@BEGIN_TEMPLATE@
  @BEGIN_Paziente@
    ([
      [[@tag_nome#FLAT@] == [Mario]]
      ?? [\'E il solito paziente Mario.]
      :: [Finalmente qualcuno che non si chiama Mario!]
    ])
  @END_Paziente@
@END_TEMPLATE@
```

Nell'esempio è mostrato un semplice esempio di costrutto *if-then-else*: viene fatto un controllo sul nome del paziente che si sta mostrando. Se il nome corrisponde a Mario verrà mostrata la scritta "È il solito paziente Mario", altrimenti verrà mostrata la scritta "Finalmente qualcuno che non si chiama Mario!".

Il mini-linguaggio Serena supporta esclusivamente le chiamate alle Serena Function, il costrutto *if-then-else* e, conseguentemente, gli operatori logici *and*, *or* e *not* e gli operatori di confronto *uguale* (`==`) e *diverso* (`!=`).

Un esempio di utilizzo delle Serena Function è invece il seguente:

```
@BEGIN_TEMPLATE@
  Ciao @FUN_GET_USER_INFO(param=username)@ !
```

[25] Per approfondimenti si veda [Coo10]

[26] Nell'esempio è utilizzato il parametro `#FLAT` che verrà descritto nel paragrafo 2.7.5.

@END_TEMPLATE@

L'esempio mostrato chiama la Serena Function `FUN_GET_USER_INFO`, tramite la quale è possibile avere alcune informazioni circa l'utente che ha effettuato il login: se ad esempio l'utente che ha effettuato il login si chiama Agnese verrà mostrata la scritta "Ciao Agnese!").

Sicuramente la Serena Function più potente è la cosiddetta `FUN_MODULE`, la cui sintassi è la seguente:

@FUN_MODULE(q=..., param1=..., param2=...)@

Tramite questa funzione è possibile chiamare da un Serena Template un altro Serena Module, ampliando notevolmente le potenzialità dei Template. Ad esempio è possibile all'interno della scheda di dettaglio di un Paziente mostrare la lista delle sue visite.

2.7.5 Serena Component

Come si è visto nella sezione precedente, nei Serena Template sono presenti dei segnaposto che indicano i dati da mostrare. La trasformazione di questi segnaposto in frammenti HTML è gestito dai cosiddetti *Serena Component*, descritti in questa sezione.

Un Serena Component è concretamente uno scheletro HTML molto semplice con alcuni segnaposto che vengono sostituiti con il dato vero e proprio da mostrare e con alcune indicazioni presenti nel Serena Interface Bean[27] (come ad esempio l'etichetta dello slot).

Un esempio di Serena Component per mostrare gli slot di tipo stringa all'interno delle schede di dettaglio è il seguente:

```
<div class="@INTERFACE_NAME@">
  <strong>@INTERFACE_LABEL@</strong>
  @DATA@
</div>
```

[27] I Serena Interface Bean descrivono gli elementi grafici di base di ogni classe e verranno analizzati nel dettaglio nella sezione 2.7.6.

Dove:

- `INTERFACE_xxx@` mostra il contenuto dell'attributo `xxx` del Serena Interface Bean corrente.
- `@DATA@` mostra il contenuto (preso dal database) dello slot corrente

Ogni metodo di ogni modulo ha un suo insieme di Serena Component, memorizzati nella directory
`app/Template/components/<nomemodulo>/<nomemetodo>`. I nomi dei singoli Serena Component è stabilito da ogni singolo metodo[28].

Qualora in un Serena Template non si voglia fare uso di un Serena Component bensì si voglia mostrare il valore grezzo di uno slot, si aggiunge all'indicatore di uno slot il parametro `#FLAT`, come nell'esempio seguente:

```
Il codice fiscale del paziente &egrave;
@tag_codice_fiscale#FLAT@
```

2.7.6 Serena Interface Bean

L'ultima parte di Serena in cui è possibile intervenire per la personalizzazione della grafica è all'interno dei Serena Interface Bean, descritti in questa sezione.

I Serena Interface Bean vengono generati automaticamente dal Serena Developer Tool con le modalità descritte nella sezione 2.1.3 e si trovano all'interno della directory `webapps/app/conf/interfaces`. Ogni classe dell'ontologia ha il suo relativo Serena Interface Bean, il cui nome è `<nomeclasse>.xml`. La struttura generica di ogni bean è la seguente:

```
<?xml version="1.0" encoding="UTF-8"?>

<bean>
  <title>@nomeclasse@</title>
```

[28] Per approfondimenti si veda [Coo10].

```
  <title_view>@titolodamostrare@</title>
  <attributes>
    <item>
      <name>ID</name>
      <label>ID</label>
    </item>
    <item>
      <name>@nomeslot@</name>
      <label>@etichettadamostrare@</label>
      <filter>@etichettadamostrarenelfiltro@</filter>
      <list>@etichettadamostrarenellalista@</list>
      <!-- ... -->
    </item>
    <!-- .. -->
  </attributes>
</bean>
```

Dove:

- Al posto di `@nomeclasse@` va inserito il nome che la classe ha nell'ontologia.
- Al posto di `@titolodamostrare@` va inserita la scritta che si vuole visualizzare nell'*header* delle pagine che gestiscono la classe.
- All'interno del tag `attributes` vanno inserite le informazioni di ogni slot, ciascuno all'interno di un tag `item`.
- Al posto di `@nomeslot@` va inserito il nome che lo slot ha nell'ontologia.
- Al posto di `@etichettadamostrare@` va inserita la scritta che va mostrata come etichetta del valore. Ad esempio per lo slot `codice_fiscale` la sua etichetta sarà "Codice fiscale", affinché quando viene richie-

sto lo slot `codice_fiscale` nella pagina verrà mostrato ad esempio[29] "Codice fiscale: CRSGNS02A41A944QT".

- Una serie di altri tag, ciascuno preposto a una sua funzione (ad esempio il tag `list` indica che lo slot va mostrato nelle liste relative alla classe data e l'etichetta da usare viene indicata nel contenuto del tag (se vuoto viene usato il contenuto del tag `label`).

Come si può facilmente dedurre, molte delle informazioni contenute nei Serena Interface Bean servono soprattutto per generare i Serena Template a partire dai Meta Template: ad esempio, nella creazione del Serena Template di una scheda di dettaglio (modulo `object`, metodo detail) di una determinata classe, gli slot da mostrare saranno quelli che nel corrispondente Serena Interface Bean contengono il tag `detail`.

I Serena Interface Bean hanno un numero elevato di parametri configurabili[30] e ogni creatore di nuovi moduli può aggiungerne altri. I parametri più importanti sono comunque gestibili attraverso il Serena Developer Tool descritto nella sezione 2.1.3.

Si conclude così tutta la panoramica su come creare e personalizzare un'applicazione Serena, dalla struttura dei dati, ai comportamenti possibili, ai permessi di accesso fino alla grafica.

Si vedrà nel prossimo capitolo come effettivamente le applicazioni Serena soddisfano le richieste che ricevono, sfruttando tutti i componenti messi a disposizione dal Serena Application Server che le ospita.

Nel prossimo capitolo verrà mostrato anche un altro strato di Serena non ancora visto: la comunicazione delle applicazioni Serena con altre applicazioni esterne[31].

[29] Qui e in altri punti della tesi è nominato un paziente Agnese Corsaro con relativo codice fiscale. Si tiene a precisare che tale paziente (come qualsiasi altra persona utilizzata negli esempi) è fittizio e assolutamente non corrispondente a persona reale.

[30] Per approfondimenti si veda [Coo10].

[31] La personalizzazione dei comportamenti relativi alla comunicazione delle applicazioni

Serena con altre applicazioni è considerata troppo approfondita per gli scopi di questo testo e verrà quindi trattata solo superficialmente nel capitolo seguente. Per approfondimenti si veda [Coo10].

Capitolo 3

STRUTTURA DEL SERENA APPLICATION SERVER

Tutte le applicazioni sviluppate con il Serena Framework necessitano, per poter essere eseguite, del Serena Application Server (di seguito chiamato Serena AS). In questo capitolo verrà presentato il Serena AS prima in modo sommario e poi andando nel dettaglio di tutti i suoi componenti.

Verranno innanzitutto descritti a volo d'angelo tutti i componenti del Serena AS, quindi verrà introdotto il protocollo di comunicazione usato tra loro. Per comprendere meglio i singoli aspetti, verrà mostrato nel dettaglio come viene eseguita una tipica richiesta HTTP. Successivamente verrà mostrato come Serena AS gestisce la comunicazione delle applicazioni Serena con altre applicazioni in rete.

3.1 Serena Application Server

La struttura del Serena Application Server è riassunta nella figura 3.1.

Come si può dedurre dalla figura, Serena AS si basa su un *servlet container* indipendente, il cui unico vincolo è supportare il protocollo *Java Servlet*[1]. Analogamente, anche il database server è indipendente a Serena AS (al mo-

[1]Per maggiori informazioni sulle Java Servlet si faccia riferimento a [Hun01].

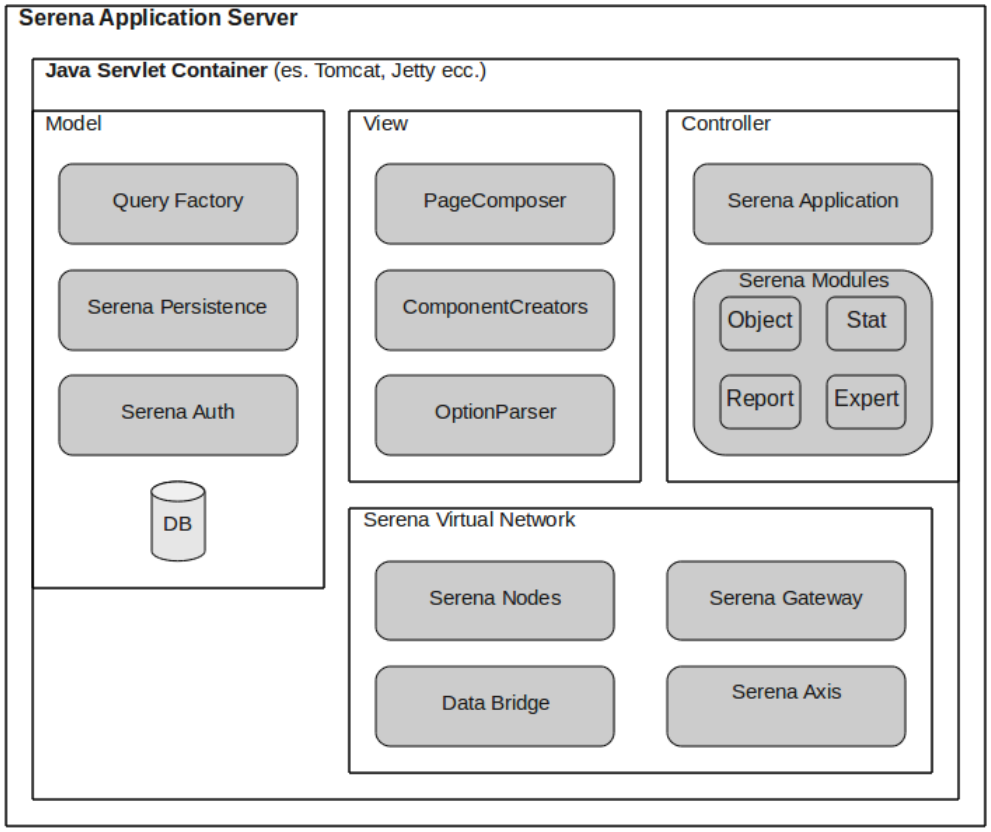

Figura 3.1: Struttura del Serena Application Server

mento della stesura di questa tesi i database supportati sono MySQL, SQL Server, Oracle e H2).

I componenti veri e propri del Serena AS sono suddivisi nella tipica logica del pattern *Model View Controller*[2]. Inoltre vi è un'ulteriore sezione, il Serena Virtual Network, dedicata alla eventuale comunicazione dell'applicazione Serena con la rete esterna, cioé con altre applicazioni Serena o con entità esterne eterogenee.

Ogni componente verrà descritto dettagliatamente nel seguito del capitolo. Qui ne viene fatto un breve riassunto:

[2]Il pattern Model View Controller è uno dei più famosi pattern architetturali. Per approfondimenti si veda [GHRJ94].

- *Serena Application*, il cuore del Serena AS nonché l'unica componente accessibile da un browser. È stato sviluppato principalmente da Andrea Frascari, Vincenzo Carnazzo, Matteo Tassetti e Andrea Pegoretti.
- I *Serena Module*, i vari plugin del Serena AS che eseguono concretamente le operazioni richieste dall'utente. I riconoscimenti agli sviluppatori sono indicati nella descrizione di ogni singolo modulo nella sezione 2.5
- *Serena Auth*, che si occupa di filtrare le richieste da fare al database in base ai privilegi dell'utente corrente. È stato sviluppato principalmente da Matteo Tassetti.
- *Serena Persistence*, che esegue concretamente le richieste al database, convertendole da XSerena a SQL. È stato sviluppato principalmente da Andrea Frascari.
- *Serena Presentation*, la parte del Serena AS che si occupa di prendere i dati richiesti e di rappresentarli graficamente all'utente. È stato sviluppato principalmente da Vincenzo Carnazzo e Matteo Tassetti.
- I *Serena Node*, che rispondono in XSerena ad eventuali richieste da parte di altre applicazioni Serena. È stato sviluppato principalmente da Vincenzo Carnazzo.
- *Serena Gateway*, che si occupa di ricevere richieste da più applicazioni Serena e smistarle a più Serena Node. È stato sviluppato principalmente da Vincenzo Carnazzo.
- *Serena Axis*, che espone dei Web Service affinché l'applicazione Serena possa dialogare con applicazioni non Serena attraverso il protocollo SOAP. È stato sviluppato principalmente da Andrea Pegoretti.

L'autore di questa tesi, come si può vedere, è dunque intervenuto attivamente su più punti dell'applicazione, in particolar modo sui componenti che si occupano della *renderizzazione* grafica dei dati e quelli che si occupano della comunicazione in rete.

Tutti i componenti (ad eccezione di Serena Axis) dialogano tra loro e verso l'esterno attraverso un protocollo di comunicazione creato ad hoc e chiamato *XSerena*, che verrà descritto nella sezione seguente.

3.2 Il protocollo XSerena

I vari componenti infrastrutturali di Serena AS dialogano tra di loro attraverso un protocollo chiamato XSerena. È un protocollo di messaggi sincroni composti in un particolare dialetto XML creato ad hoc. Tale protocollo permette la tipica flessibilità dell'XML, che ben si presta a rappresentazioni ad albero dei dati, evitando l'eccessiva complessità di altri linguaggi noti, come ad esempio SOAP[3].

Un tipico scambio di messaggi XSerena è il seguente (il primo è una richiesta, il secondo è una risposta):

```
<serena action="request">
  ...
  <service name="service-x">
    ...
  </service>
</serena>

<serena action="response">
  <service name="service-x">
    ...
  </service>
  <metadata>
    <result>...</result>
    ...
  </metadata>
</serena>
```

[3]Per un approfondimento sulle differenze tra XSerena e SOAP si veda 5.2.

Dove:

- `serena` è il tag principale, il cui attributo `action` indica se è una richiesta o una risposta.
- `service` è il tag che contiene la richiesta vera e propria (che viene ricopiata nella risposta). L'attributo `name` contiene il nome del servizio richiesto. I servizi disponibili dipendono dal componente che si sta interrogando e verranno quindi esposti dettagliatamente più avanti in questo capitolo.
- `metadata` è il tag che contiene eventuali metainformazioni non strettamente relative alla struttura ontologica dei dati. Ad esempio contiene il tag `result` che indica se la richiesta è andata a buon fine o meno.

All'interno di tale struttura è possibile inserire dati eterogenei (dipendenti dai componenti che si sta utilizzando e dai dati che si stanno gestendo) ma tutti seguendo il modello ontologico.

Ad esempio, un documento XSerena che rispondesse con successo al servizio *get* (l'equivalente delle `SELECT` SQL) e che contesse i dati rappresentati nella figura 2.1 sarebbe composto così:

```
<serena action="response">
  <service name="get">
    <Paziente>
      <codice_fiscale>CRSGNS02A41A944Q</codice_fiscale>
      <medico_di_famiglia>
        <Medico>
          <nome>Marco</nome>
          <cognome>Lelli</cognome>
        </Medico>
      </medico_di_famiglia>
    </Paziente>
  </service>
```

```
  <metadata>
    <result>1</result>
  </metadata>
</serena>
```

Quasi tutte le comunicazioni tra i componenti Serena, allo stato attuale, avviene direttamente attraverso codice Java. L'uso del protocollo XSerena per trasmettere richieste e dati tra di loro renderebbe comunque possibile, qualora in futuro si ritenesse necessario, un tipo di comunicazione diverso tra di loro (ad esempio tramite HTTP).

I componenti che fanno eccezione, che cioè possono essere contattate dall'esterno, sono Serena Application e i Serena Node.

La comunicazione con le servlet di Serena Application è una normale comunicazione HTTP, senza l'uso di XSerena, e con l'aggiunta di una gestione dello stato (esiste ad esempio una variabile di sessione che indica quale utente è al momento *loggato*, informazione che permette nel sistema la gestione dei permessi).

Lo stato è memorizzato utilizzando le API `HttpSession`: tali API sono fornite da ogni *servlet container* (purché rispetti lo standard Java Servlets) e rendono trasparente ai programmatori le modalità di creazione di tali sessioni. I principali *servlet container*, come ad esempio Tomcat, memorizzano i dati di sessione attraverso *cookies* oppure, qualora i *cookies* non siano supportati dal client, al loro interno, identificandoli con un identificativo che il client fornirà tra i parametri passati nel *query string*.

La comunicazione con i Serena Node avviene invece usando messaggi XSerena sopra al protocollo HTTP, senza alcun stato[4].

3.3 Il ciclo di vita tipico di una richiesta

Per poter comprendere meglio Serena AS e i suoi componenti, verrà ora analizzato il ciclo di vita di una tipica richiesta HTTP a un'applicazione

[4] I Serena Node verranno approfonditi nella sezione 3.9.1.

Serena, analizzando poi nel dettaglio ogni componente coinvolto.

Il ciclo di vita della richiesta è il seguente:

1. L'utente tramite browser richiede un determinato URL a cui risponde l'applicazione Serena.

2. Il servlet container (Tomcat, Jetty o simile) installato nel server identifica la chiamata e la passa a Serena Application.

3. Serena Application analizza la richiesta e identifica quali Serena Module sono coinvolti nella richiesta. Chiama quindi ogni Serena Module richiesto.

4. Ogni Serena Module chiamato esegue le operazioni richieste e crea una risposta sotto forma di un frammento HTML.

5. Serena Application mette insieme le risposte di tutti i Serena Module, le posiziona dove opportuno all'interno della pagina, aggiunge le parti statiche della parte e ottiene il risultato finale, leggibile dall'utente.

6. Serena Application ritorna la risposta al servlet container

7. Il servlet container ritorna la risposta al browser

Tutto il processo è riassunto nel *sequence diagram* della figura 3.2[5].

Nel resto del capitolo viene mostrato nel dettaglio la struttura e il comportamento di ogni singolo componente coinvolto in questo ciclo di vita.

[5]Nel diagramma è mostrata tra le azione la creazione di un nuovo Meta-Ambiente. Ciò è vero solo per la prima richiesta. Nelle richieste successive, salvo cambio di Meta-Ambiente, la classe `MetaEnvironment` e la lista dei Serena Module attivi viene reperita dallo stato corrente.

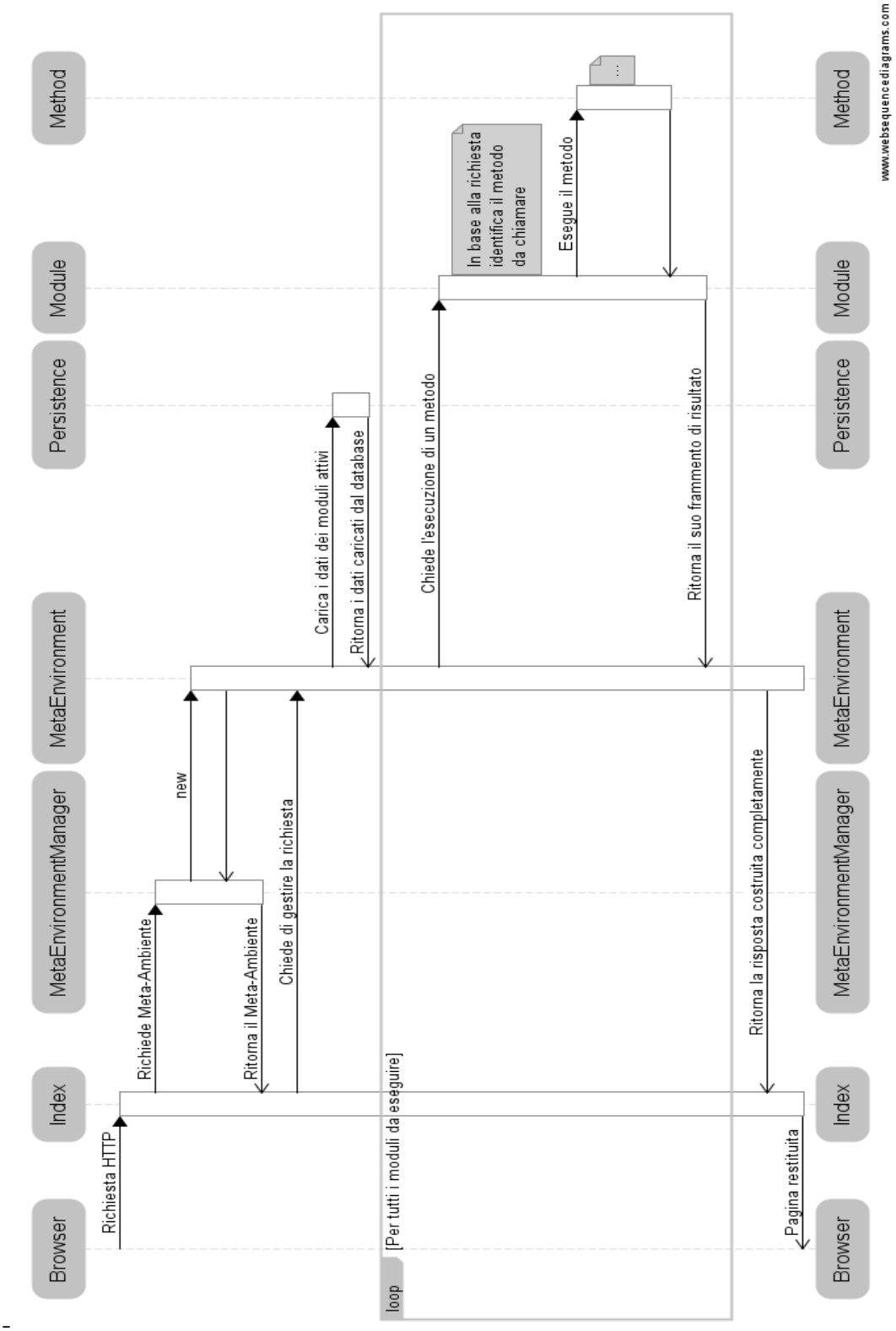

Figura 3.2: *Sequence diagram* del ciclo di vita tipico di una richiesta

3.4 Serena Application

Serena Application rappresenta il cuore di Serena. Si occupa di mostrare verso l'esterno le servlet principali del sistema, contattabili da un normale browser.

Le servlet esposte sono le seguenti:

- `Index`. È la servlet principale e predefinita. Il suo comportamento cambia in base ai parametri che le vengono passati.
- `ModuleIndex`. Ha un comportamento simile a `Index`, con la particolarità che contatta direttamente un solo Serena Module. È utile per chiamate dove non è necessario ricostruire l'intera grafica della pagina, come ad esempio le chiamate AJAX.
- `Attachment`. È la servlet contattata per ottenere un determinato allegato, cioé un elemento non HTML (es. un immagine, un video ecc.).
- `Rss`. Per le applicazioni Serena che li supportano, crea un *feed RSS* dei contenuti dell'applicazione.
- `Image`. Mostra l'anteprima di un allegato di tipo immagine.

Le servlet `Attachment`, `Rss` ed `Image` non sono altro che specializzazioni di `ModuleIndex`, con parametri preimpostati. La loro esistenza è giustificata solo dall'esigenza di avere in alcuni casi degli URL più leggibili. `ModuleIndex` a sua volta è un caso particolare di `Index`, pertanto per semplicità di seguito verranno considerate solo le richieste alla servlet `Index`.

3.4.1 I meta-ambienti

Per poter elaborare le richieste HTTP ricevute e costruire la grafica della pagina da restituire all'utente, Serena Application ha bisogno di sapere:

- quali sono i Serena Module attualmente attivi (in base ai permessi dell'utente corrente e allo stato corrente dell'applicazione)
- qual è il tema grafico dell'applicazione nel suo stato corrente

A contenere queste informazioni sono i cosiddetti Meta-Ambienti, descritti già nella sezione 2.7.1. Ogni Meta-Ambiente è descritto nel sistema attraverso:

- Un'istanza della classe ontologica `_system_meta_environment`, conservata nel database e contenente tutte le informazioni utili (il nome del Meta-Ambiente, i moduli ad esso associati ecc.)
- Un'istanza della classe Java `MetaEnvironment`.

Internamente i Meta-Ambienti sono gestiti dal cosiddetto `MetaEnvironmentManager`: a ogni richiesta ricevuta, la servlet `Index` chiede al `MetaEnvironmentManager` qual è il Meta Ambiente corrente e a quest'ultimo chiede di espletare la richiesta ricevuta. L'istanza corrente di `MetaEnvironment` chiama tutti i Serena Module da chiamare automaticamente e il Serena Module chiesto esplicitamente.

La struttura e il comportamento generico dei Serena Module è mostrato nella sezione seguente.

3.5 I Serena Module

I Serena Module sono dei plugin installabili in ogni applicazione Serena. Sono stati descritti ampiamente nella sezione 2.5. Concretamente ogni Serena Module è composto da:

- Un'istanza della classe ontologica `_system_module` conservata nel database e contenente tutte le informazioni utili.
- Una classe Java che implementa l'interfaccia `SerenaModule` e che contiene il codice da eseguire nel caso in cui il modulo venga invocato.

L'istanza ontologica contiene le seguenti informazioni:

- Il nome del modulo. Si tratta del nome concreto del modulo, cioé ciò che poi bisogna inserire nella *query string* per identificare il modulo.
- Il Meta-Ambiente a cui appartiene
- La posizione all'interno del Meta-Ambiente, Se ad esempio la posizione indicata è 2, il frammento di HTML di tale modulo verrà inserito al posto del segnaposto `@MOD_2@`.
- L'ordine: questo campo verrà preso in considerazione per stabilire l'ordine di visualizzazione nel caso in cui ci siano più moduli nella stessa posizione.
- La modalità di attivazione. Normalmente un modulo viene attivato su esplicita richiesta dell'utente, ma può anche essere impostato tramite questo campo che venga attivato ad ogni richiesta oppure solo in home page (cioé quando non viene richiesto nessun altro modulo).
- Il nome del metodo di default: è il metodo che viene chiamato nel caso in cui si è impostato di attivare il modulo ad ogni richiesta oppure solo in home page.
- I parametri di default. Sono i parametri passati sempre e comunque al metodo chiamato. È utile ad esempio per indicare le informazioni utili per identificare l'oggetto che si vuole mostrare in home page.
- Il nome e il package della classe Java associata al modulo.

La classe Java ha alcuni vincoli implementativi:

- Deve avere lo stesso nome e package indicati in `_system_module`
- Deve implementare l'interfaccia `SerenaModule` (contenuta nel core di Serena)

- Deve indicare, tramite la funzione `getName` il suo nome di default (ad esempio object)
- Deve implementare la funzione `doMethod`, che è quella chiamata da Serena Application quando vuole lanciare il modulo. Generalmente si occupa solo di identificare il metodo richiesto e passare la gestione al metodo (un'ulteriore classe Java che implementa l'interfaccia `SerenaMethod`).

Solitamente il comportamento di un Serena Module, appena invocato, è il seguente:

1. Analizza la parte di richiesta che gli compete e identifica il metodo da usare.
2. Chiama il metodo identificato. Il quale:
   (a) Se necessario, converte la richiesta HTTP in una richiesta in XSerena per il database (di lettura e/o di scrittura).
   (b) Chiede a Serena Auth di filtrare la richiesta al database in base ai permessi dell'utente richiedente. In alcuni casi questa fase può essere saltata (ad esempio all'interno del modulo `Login`). Per le implicazioni relative alla sicurezza si veda la sezione 4.2.
   (c) Chiede a Serena Persistence di eseguire la richiesta.
   (d) Legge la risposta ricevuta da Serena Persistence (ed eventualmente la elabora ulteriormente).
   (e) Chiede a Serena Presentation di rendere graficamente presentabile la risposta
   (f) Ritorna al modulo la sua risposta in HTML
3. Ritorna a Serena Application la sua parte di risposta in HTML

Tutto il processo è riassunto nel *sequence diagram* della figura 3.3.

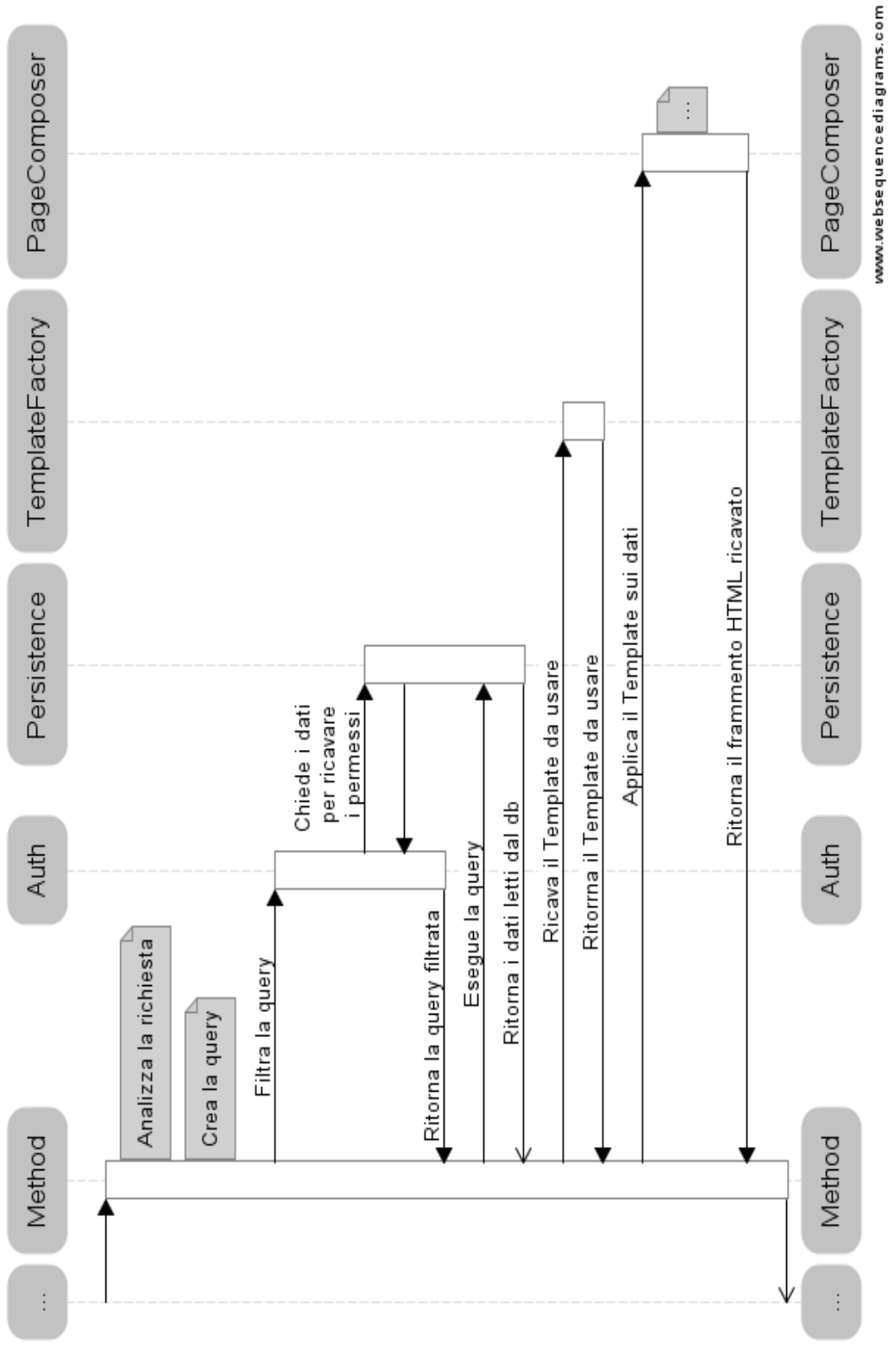

Figura 3.3: *Sequence diagram* di una tipica esecuzione di un Serena Module

Come si può facilmente notare, quasi tutti i Serena Module necessitano di un accesso al database durante il loro funzionamento. Tale accesso avviene tramite messaggi XSerena inviati al componente Serena Persistence, descritto nella sezione seguente.

3.6 Serena Persistence

Quasi tutti i componenti del Serena AS necessitano di un'interazione con un database per poter funzionare: Serena Application per ricavare i dati relativi al Meta-Ambiente, Serena Auth per ricavare i dati relativi ai permessi, molti dei Serena Module per poter gestire i dati ecc.

L'interazione tra il resto dell'infrastruttura e i database fisici è gestita da Serena Persistence. Serena Persistence si occupa di ricevere richieste espresse in XSerena (quindi in un linguaggio neutro) e di tradurle nel dialetto SQL specifico del database fisico installato. Al momento della stesura di questo testo i database supportati sono SQL Server, MySQL, Oracle, Access e H2.

Il comportamento specifico del Serena Persistence dipende da quanto impostato nel file `config_persistence.xml` e nei *Serena Entity Bean*, descritti rispettivamente nelle sezioni 2.1.4 e 2.6.

Serena Persistence supporta due tipi di richiesta: le richieste di tipo *get*, per leggere dati dal database, e le richieste di tipo *set*, per scrivere dati nel database.

3.6.1 Richieste di tipo *get*

Le richieste di tipo *get* servono per leggere i dati dal database. Sono nella forma seguente:

```
<serena action="request">
  <service name="get">
    <IstanzaPrincipale operation="select" target="?">
      <condition>
```

```
        ...
      </condition>
      <slot_ground1 />
      <slot_ground2 />
      ...
      <slot_relazionale1>
        <IstanzaDiRelazione operation="select" target="?">
          ...
        </IstanzaDiRelazione>
      </slot_relazionale1>

      <slot_relazionale2 />
      ...
    </IstanzaPrincipale>
  </service>
</serena>
```

La struttura principale della richiesta è quella tipica di un messaggio XSerena, con le seguenti particolarità:

- L'attributo `name` del tag `service` indica che la richiesta è una `get`.
- Come figlio del tag `service` va inserita l'istanza principale della richiesta, cioé l'istanza da cui far partire la richiesta.
- Ogni istanza ha un attributo `operation` che indica il tipo di operazione da eseguire su quella istanza. Per le richieste di tipo *get* l'unica operazione possibile è *select*.
- Ogni istanza ha un attributo `target` che può avere come valori: `?` (vanno chiesti al database solo gli slot dell'istanza esplicitamente indicati all'interno della richiesta) oppure `*` (vanno chiesto tutti gli slot dell'istanza).

- In caso l'attributo `target` sia valorizzato con `?`, all'interno dell'istanza vanno inseriti tag con il nome degli slot che si vuole richiedere.
- Per gli slot ground vanno inseriti tag vuoti.
- Per gli slot di relazione è possibile inserire il tag con all'interno un'ulteriore frammento XML relativo all'istanza di relazione la cui sintassi è uguale a quella relativa all'istanza principale.
- Per gli slot di relazione è anche possibile inserire tag vuoti: equivale a una richiesta dell'istanza di relazione con `target=''*''`.
- Ogni istanza può avere anche un tag `condition` dove indicare (secondo la consueta sintassi dei messaggi XSerena) eventuali condizioni di filtraggio della richiesta.

Per chiarire meglio la sintassi, ecco come chiedere tutti i pazienti il cui cognome è Rossi. Di tali pazienti viene richiesto il codice fiscale, il medico di famiglia e la prima visita. Del medico viene chiesto il nome e il cognome. Della prima visita vengono chieste tutte le informazioni.

```
<serena action="request">
  <service name="get">
    <Paziente operation="select" target="?">
      <condition>
        <cognome>Rossi</cognome>
      </condition>
      <codice_fiscale />
      <medico_di_famiglia>
        <Medico operation="select" target="?">
          <nome />
          <cognome />
        </Medico>
      </medico_di_famiglia>
      <prima_visita />
```

```
    </Paziente>
  </service>
  <metadata>
    <result>1</result>
  </metadata>
</serena>
```

Per rispondere alla richiesta Serena Persistence navigherà all'interno del XSerena e chiederà al database via via ciò che gli serve. In caso incontri slot di relazione, chiederà al database la *foreign key* e successivamente farà ulteriori chiamate al database per chiedere le informazioni relative alle istanze di relazione.

Dunque per rispondere alla precedente richiesta, ammettendo che le *foreign key* siano `ID_medico_di_famiglia` e `ID_prima_visita` e che contengano rispettivamente come valori 1 e 2, le chiamate SQL concrete saranno:

```
SELECT codice_fiscale, ID_medico_di_famiglia,
       ID_prima_visita FROM Paziente;
SELECT nome, cognome FROM Medico WHERE ID=1;
SELECT * FROM Visita WHERE ID=2;
```

Una volta ricevuta risposta dal database, Serena Persistence rimetterà insieme i pezzi e tornerà una risposta simile alla seguente:

```
<serena action="response">
  <service name="get">
    <Paziente>
      <codice_fiscale>
        CRSGNS02A41A944Q
      </codice_fiscale>
      <medico_di_famiglia>
        <Medico>
          <nome>Marco</nome>
          <cognome>Lelli</cognome>
```

```
        </Medico>
      </medico_di_famiglia>
      <prima_visita>
        <Visita>
          <data>02/09/2010</data>
          <tipologia_visita>
            Controllo ordinario
          </tipologia_visita>
          ...
        </Visita>
      </prima_visita>
    </Paziente>
    <Paziente>
      <codice_fiscale>CRN...</codice_fiscale>
      <medico_di_famiglia>
        <Medico>
          <nome>Enrico</nome>
          <cognome>Signori</cognome>
        </Medico>
      </medico_di_famiglia>
      <prima_visita>
        <Visita>
          <data>01/10/2010</data>
          <tipologia_visita>
            Controllo ordinario
          </tipologia_visita>
          ...
        </Visita>
      </prima_visita>
    </Paziente>
  </service>
```

```
  <metadata>
    <result dimension="2">1</result>
  </metadata>
</serena>
```

Si noti che naturalmente possono esistere più record in risposta. Il tag `result` conterrà il valore 1 per indicare che l'operazione è andata a buon fine e avrà un attributo `dimension` valorizzato a 2 per indicare che sono stati trovati due pazienti.

Le richieste di tipo *get* supportate da Serena Persistence possono essere molto più varie e dettagliate di quelle qui descritte: sono gestiti ad esempio l'ordinamento, la possibilità di effettuare ricerche tramite operatore `like`, limitare il numero di risultati ottenuti, sia in lunghezza che in profondità. Esula dagli scopi di questo testo entrare troppo nel dettaglio su tali informazioni[6].

3.6.2 Richieste di tipo *set*

Le richieste di tipo *set* servono per scrivere i dati nel database. Sono nella forma seguente:

```
<serena action="request">
  <service name="set">
    <IstanzaPrincipale operation="update">
      <condition>
        ...
      </condition>
      <slot_ground1>valore1</slot_ground1>
      <slot_ground2>valore2</slot_ground2>
      ...
      <slot_relazionale1>
        <IstanzaDiRelazione1 operation="insert">
```

[6]Per approfondimenti si rimanda alla documentazione ufficiale di Serena [Coo10].

```
            <ID>1</ID>
          </IstanzaDiRelazione1>
        </slot_relazionale1>
        <slot_relazionale2>
          <IstanzaDiRelazione2 operation="insert">
            ...
          </IstanzaDiRelazione2>
        </slot_relazionale2>
      </IstanzaPrincipale>
    </service>
</serena>
```

Come si può vedere, la struttura principale è molto simile alle richieste di tipo *get* viste nella sezione precedente. Si rimanda a tale sezione per i concetti generali. Si evidenziano qui invece le differenze rispetto alle richieste già viste.

- Il tag `name` indica che la richiesta è una `set`.
- Ogni istanza ha un attributo `operation` che indica il tipo di operazione da eseguire su quella istanza. Le richieste supportate sono: `insert` (inserimento di una nuova istanza); `update` (modifica di un'istanza esistente); `delete` (cancellazione di un'istanza esistente).
- Negli slot ground va inserito il valore che gli si vuol dare.
- Negli slot di relazione vanno inserite anche le informazioni relative alle istanze di relazione.
- Nelle istanze con l'attributo `operation` valorizzato con `update` o `delete` è possibile inserito il tag `condition` che aggiunge una condizione all'operazione da effettuare.

Ecco ad esempio come prendere il Paziente con ID 1, modificargli il cognome e aggiungergli il medico di famiglia.

```
<serena action="request">
  <service name="get">
    <Paziente operation="update">
      <condition>
        <ID>1</ID>
      </condition>
      <cognome>Rossi</cognome>
      <medico_di_famiglia>
        <Medico operation="insert">
          <nome>Marco</nome>
          <cognome>Lelli>
        </Medico>
      </medico_di_famiglia>
    </Paziente>
  </service>
</serena>
```

Per eseguire la richiesta Serena Persistence navigherà all'interno del XSerena e inizierà le operazioni a partire dalle istanze più in profondità. Per ognuna effettuerà le operazioni richieste e si terrà l'ID del record utilizzato, per poterlo poi riutilizzare per la *foreign key* dell'istanza padre.

Dunque per esegure la precedente richiesta, ammettendo che la *foreign key* sia `ID_medico_di_famiglia` e l'ID di inserimento del nuovo medico sia 2, le chiamate SQL concrete saranno:

```
INSERT INTO Medico (nome, cognome)
        VALUES("Marco", "Lelli");
UPDATE Paziente SET cognome="Rossi",
        ID_medico_di_famiglia=2 WHERE ID=1
```

Le risposte che Serena Persistence dà a tali richieste sono molto più semplici di quelle di tipo *get*. Fondamentalmente le risposte indicano se l'operazione è andata a buon fine e, in caso di *insert*, danno l'ID della nuova istanza inserita. Ecco un esempio:

```
<serena action="response">
  <service name="set">
  </service>
  <metadata>
    <result new_id="2">1</result>
  </metadata>
</serena>
```

Le richieste di tipo *set* supportate da Serena Persistence possono essere molto più varie e dettagliate di quelle qui descritte e, come è facile immaginare, gli scenari possibili sono molto più variegati (inserimento di record già esistenti, creazione di relazioni tra dati già presenti ecc.). Esula dagli scopi di questo testo entrare troppo nel dettaglio su tali informazioni[7].

Raramente comunque le componenti di Serena utilizzano Serena Persistence direttamente: ogni richiesta XSerena va prima modificata affinché vengano rispettati i permessi di utilizzo dell'utente che le sta eseguendo. Perciò ogni richiesta XSerena, prima di essere inviata a Serena Persistence, viene inviata a Serena Auth, descritto nella sezione seguente.

3.7 Serena Auth

Per poter far sì che funzioni il sistema di gestione dei permessi descritto nella sezione 2.4, è necessario che ogni richiesta che i Serena Module vogliono inviare al Serena Persistence sia prima filtrata[8].

Serena Auth è il componente del Serena AS preposto al filtraggio delle richieste. Esso prende in input le richieste da filtrare e ritorna le richieste leggermente modificate affinché i risultati ottenuti siano adeguati ai permessi dell'utente corrente cioé l'utente inserito nella corrente sessione HTTP. Qua-

[7] Per approfondimenti si rimanda alla documentazione ufficiale di Serena [Coo10].

[8] Ci sono comunque rare eccezioni: ad esempio la richiesta del modulo Login per controllare se l'username e la password inseriti dall'utente per loggarsi sono corretti.

lora in sessione non fosse presente nessun utente, viene utilizzato l'utente fittizio *everyone*.

Serena Auth ogni qualvolta viene coinvolto interroga il database tramite Serena Persistence e ottiene i dati necessari a ricavare i permessi dell'utente: i gruppi a cui l'utente appartiene, i suoi colleghi, i permessi di classe e di istanza relativi ai dati su cui si sta lavorando e gli utenti creatori di tali istanze.

Concretamente il filtraggio è una modifica della richiesta XSerena secondo i seguenti criteri:

- Se la richiesta è di inserimento/modifica/cancellazione su un'istanza e l'utente non ha i permessi per eseguirla, Serena Auth dà errore.
- Se la richiesta è di lettura, viene modificata aggiungendo ulteriori condizioni affinché vengano visualizzate solo le istanze per le quali l'utente ha i permessi.

Ad esempio supponiamo che l'utente corrente si chiami *vcarnazzo* e abbia i permessi di lettura sulle istanze di classe *Paziente* purché siano state create da sè stesso. Supponiamo anche che Serena Auth riceva la seguente richiesta da filtrare:

```
<serena action="request">
  <service name="get">
    <Paziente operation="select" target="?">
      <codice_fiscale />
    </Paziente>
  </service>
</serena>
```

In questo scenario Serena Auth modificherà la richiesta come segue:

```
<serena action="request">
  <service name="get">
    <Paziente operation="select" target="?">
```

```
        <condition>
          <creation_user>vcarnazzo</creation_user>
        </condition>
        <codice_fiscale />
      </Paziente>
    </service>
</serena>
```

Abbiamo visto così come vengono richiesti i dati al database. Per completare il ciclo di vita di una richiesta a un Serena AS resta da mostrare come tali dati vengono trasformati affinché il Serena Method interrogato possa creare il suo frammento di codice HTML. A ciò è preposto il componente Serena Presentation.

3.8 Serena Presentation

Per poter trasformare i propri dati (espressi in XSerena) in codice HTML ogni modulo fa uso del componente Serena Presentation.

Per poter comprendere come lavora Serena Presentation, bisogna avere chiari i concetti di Serena Template, Serena Meta-Template e di Serena Component, descritti nelle sezioni 2.7.2 e 2.7.5.

Serena Presentation è suddiviso a sua volta nei seguenti componenti[9]:

- *Page Composer*. È il componente principale ed è l'unica parte di Serena Presentation richiamabile dal resto di Serena AS. Si occupa di prendere i dati dall'esterno e coordinare le varie parti di Serena Presentation per ottenere il risultato finale da ritornare al richiedente.

[9]Nell'elenco manca un ulteriore componente chiamato *Query Factory*, che opera in modo inverso del consueto: legge un Serena Template e in base ad esso crea la query da inviare a Serena Persistence. Per semplicità non viene qui approfondito il suo funzionamento. Si rimanda per ciò a [Coo10].

- *Template Factory.* Viene richiamato dal Page Composer all'inizio della sua esecuzione. Si occupa di individuare il Serena Template da usare o, qualora non esistesse, di crearlo a partire da un Serena Meta-Template.

- *Function Parser.* Viene richiamato dal Page Composer all'inizio e alla fine del *parsing* di un Serena Template. Analizza un Serena Template e identifica eventuali Serena Function (rispettivamente di *preparsing* e di *postparsing*), li richiama e inserisce nel Template gli eventuali valori restituiti.

- *Component Creator.* Viene richiamato dal Page Composer durante il *parsing* delle istanze e degli slot. Per ogni istanza, slot ground e slot di relazione, identifica il Serena Component da usare. Esiste un *Component Creator* generico che crea un determinato Serena Component per ogni tipo di dato (uno per le istanze, uno per gli slot di relazione 1-N, uno per quelli M-N, uno per gli slot ground di tipo stringa, uno per quelli di tipo data ecc.). Potenzialmente ogni Serena Module può avere un suo specifico Component Creator che ha comportamenti diversi rispetto a quello generico.

- *Serena Component.* Vengono richiamati dal Page Composer ogni volta che viene trovato un dato (sia un'istanza, slot di relazione o slot ground). Si occupano di creare un frammento HTML per ogni dato trovato. Ad esempio, per una scheda di dettaglio, esisterà un Serena Component che creerà una *form* per un'istanza, una *combobox* per uno slot relazionale, un campo di testo per uno slot ground di tipo stringa ecc.

- *Option Parser.* Viene richiamato dal Page Composer subito dopo aver usato tutti i Serena Component. Si occupa di effettuare il *parsing* del mini-lingaggio di Serena.

Una volta mostrati tutti i componenti in gioco, ecco come interagiscono nella sequenza che porta alla trasformazione dei dati in HTML:

1. Un Serena Method passa al Page Composer i dati da mostrare, il suo nome e il nome del suo modulo (necessari per ricavare il Serena Template da usare).

2. Il Page Composer richiede al Template Factory il Serena Template da usare. Il Template Factory ricava il Serena Template da usare e lo ritorna.

3. Il Page Composer chiede al Function Parser di analizzare il Serena Template. Il Function Parser analizza il Serena Template ed identifica le Serena Function in preparsing. Chiama ciascuna di esse e sostituisce nel template la loro chiamata con il valore ritornato. Alla fine ritorna il Serena Template modificato al Page Composer.

4. Il Page Composer analizza il Serena Template alla ricerca di tag relativi ai dati. Per ogni tag che indica l'inizio di un'istanza, l'inizio di una relazione o uno slot ground:

   (a) Il Page Composer chiede al Component Creator il Serena Component da usare.

   (b) Il Component Creator analizza il tag: controlla se è relativo a un'istanza, a uno slot di relazione o a uno slot ground e, in caso di slot, di che tipo di slot si tratta (reperendo le informazioni dal Serena Entity Bean della classe opportuna). In base a tali informazioni ritorna il Serena Component corretto.

   (c) Il Page Composer passa al Serena Component ottenuto i dati che deve mostrare.

   (d) Il Serena Component carica il suo scheletro HTML, lo riempie con i dati che ha ricevuto e ritorna il micro-frammento di HTML relativo.

   (e) Il Page Composer sostituisce al tag il micro-frammento di HTML ottenuto e prosegue al prossimo tag.

5. Il Page Composer passa il Serena Template all'Option Parser. L'Option Parser analizza il Serena Template alla ricerca di tutti i frammenti di mini-linguaggio. Ogni frammento lo esegue e sostituisce il frammento con il risultato ottenuto. Alla fine ritorna il Serena Tempalte modificato al Page Composer.

6. Il Page Composer chiede al Function Parser di analizzare nuovamente il Serena Template. Il Function Parser analizza il Serena Template ed identifica le Serena Function in postparsing. Chiama ciascuna di esse e sostituisce nel template la loro chiamata con il valore ritornato. Alla fine ritorna il Serena Template modificato al Page Composer.

7. Il Page Composer a questo punto ha un Serena Template senza più tag e con solo HTML.

Tutto il processo è riassunto nel *sequence diagram* della figura 3.4.

Per chiarire meglio, qui di seguito viene mostrato un frammento di Serena Template di esempio e come nelle varie fasi venga modificato fino ad arrivare al frammento HTML finale.

Serena Template iniziale:

```
@BEGIN_TEMPLATE@
  <div class="title">
    <h3>Scheda di dettaglio di un paziente</h3>
  </div>
  <div class="content">
    <!-- Inizio dei dati -->
    @BEGIN_Paziente@
      @tag_codice_fiscale@

      <hr />

      <h4>Medico di famiglia:</h4>
      @BEGIN_medico_di_famiglia@
```

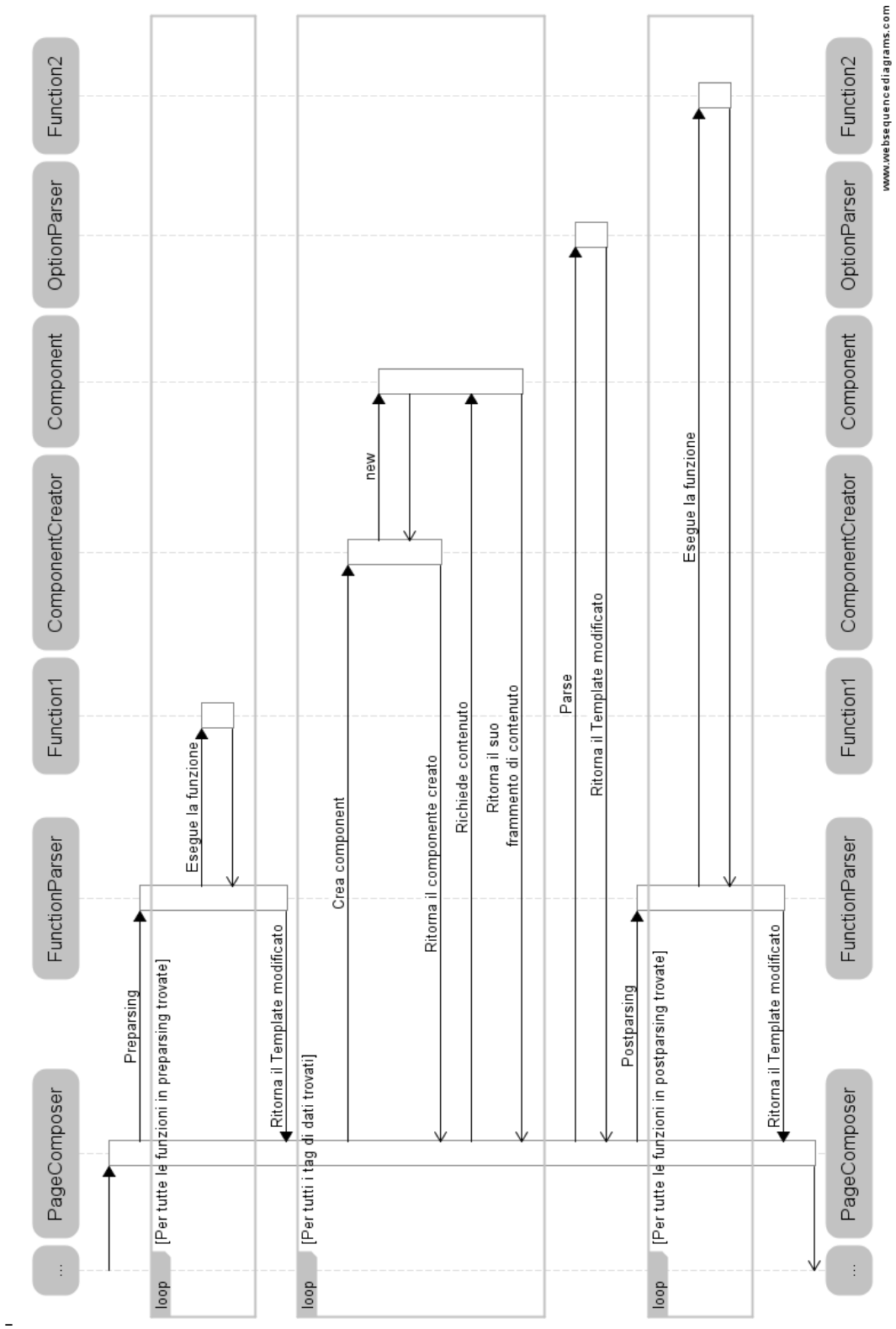

Figura 3.4: *Sequence diagram* della conversione dei dati grezzi in HTML

```
        @BEGIN_Medico@
          @tag_cognome@
          @tag_nome@
        @END_Medico@
      @END_medico_di_famiglia@

      <hr />
      <p>
        La prima visita &egrave; stata effettuata il
        @tag_prima_visita.Visita.data#FLAT@
      </p>

      <!-- Mini linguaggio con funzione
           in preparsing nella condition -->
      ([
        [ [@FUN_IS_USER_IN(group=admin)@] == [true] ]
        ?? [
          <h3>Modifica del paziente</h3>
          <!-- Funzione in postparsing -->
          @FUN_MODULE(q=object/detail_edit,p=...)@
        ]
      ])
    @END_Paziente@
  </div>
@END_TEMPLATE@
```

Dopo il preparsing del Function Parser viene risolta la chiamata a `FUN_IS_USER_IN` (funzione che torna `true` se l'utente corrente fa parte del gruppo indicato; falso altrimenti):

```
@BEGIN_TEMPLATE@
  <div class="title">
    <h3>Scheda di dettaglio di un paziente</h3>
```

```
</div>
<div class="content">
  <!-- Inizio dei dati -->
  @BEGIN_Paziente@
    @tag_codice_fiscale@

    <hr />

    <h4>Medico di famiglia:</h4>
    @BEGIN_medico_di_famiglia@
      @BEGIN_Medico@
        @tag_cognome@
        @tag_nome@
      @END_Medico@
    @END_medico_di_famiglia@

    <hr />
    <p>
       La prima visita &egrave; stata effettuata il
       @tag_prima_visita.Visita.data#FLAT@</p>

    <!-- Mini linguaggio con funzione
         in preparsing nella condition -->
    ([
      [ [true] == [true] ]
      ?? [
        <h3>Modifica del paziente</h3>
        <!-- Funzione in postparsing -->
        @FUN_MODULE(q=object/detail_edit,p=...)@
      ]
    ])
```

```
    @END_Paziente@
  </div>
@END_TEMPLATE@
```

Dopo il parsing tramite Component Creator tutta la parte relativa ai dati diventa HTML puro (da notare che il dato relativo alla data della visita, avendo il parametro `#FLAT` non usa alcun Serena Component):

```
@BEGIN_TEMPLATE@
  <div class="title">
    <h3>Scheda di dettaglio di un paziente</h3>
  </div>
  <div class="content">
    <!-- Inizio dei dati -->
    <div class="Paziente">
      <p>
        <strong>Codice fiscale:</strong>
        CRSGNS02A41A944Q
      </p>

      <hr />

      <h4>Medico di famiglia:</h4>
      <div class="medico_di_famiglia">
        <div class="Medico">
          <p><strong>Cognome:</strong> Lelli</p>
          <p><strong>Nome:</strong> Marco</p>
        </div>
      </div>
      <hr />
      <p>
         La prima visita &egrave; stata effettuata il
         12 febbraio 2010</p>
```

```
        <!-- Mini linguaggio con funzione in
             preparsing nella condition -->
        ([
          [ [true] == [true] ]
          ?? [
            <h3>Modifica del paziente</h3>
            <!-- Funzione in postparsing -->
            @FUN_MODULE(q=object/detail_edit,p=...)@
          ]
        ])
      </div>
    </div>
@END_TEMPLATE@
```

Dopo il parsing dell'Option Parser viene mostrato il frammento all'interno del `then` poiché la condizione risulta verificata:

```
@BEGIN_TEMPLATE@
  <div class="title">
    <h3>Scheda di dettaglio di un paziente</h3>
  </div>
  <div class="content">
    <!-- Inizio dei dati -->
    <div class="Paziente">
      <p>
        <strong>Codice fiscale:</strong>
        CRSGNS02A41A944Q
      </p>

      <hr />

      <h4>Medico di famiglia:</h4>
```

```
        <div class="medico_di_famiglia">
          <div class="Medico">
            <p><strong>Cognome:</strong> Lelli</p>
            <p><strong>Nome:</strong> Marco</p>
          </div>
        </div>
        <hr />
        <p>
           La prima visita &egrave; stata effettuata il
           12 febbraio 2010</p>

        <!-- Mini linguaggio con funzione in
             preparsing nella condition -->
        <h3>Modifica del paziente</h3>
        <!-- Funzione in postparsing -->
        @FUN_MODULE(q=object/detail_edit,p=...)@
      </div>
    </div>
@END_TEMPLATE@
```

Dopo il postparsing del Function Parser viene risolta la chiamata a `FUN_MODULE` (funzione che chiama un ulteriore metodo di un modulo, il cui contenuto qui viene ristretto per semplicità):

```
@BEGIN_TEMPLATE@
  <div class="title">
    <h3>Scheda di dettaglio di un paziente</h3>
  </div>
  <div class="content">
    <!-- Inizio dei dati -->
    <div class="Paziente">
      <p>
        <strong>Codice fiscale:</strong>
```

```
        CRSGNS02A41A944Q
      </p>

      <hr />

      <h4>Medico di famiglia:</h4>
      <div class="medico_di_famiglia">
        <div class="Medico">
          <p><strong>Cognome:</strong> Lelli</p>
          <p><strong>Nome:</strong> Marco</p>
        </div>
      </div>
      <hr />
      <p>
         La prima visita &egrave; stata effettuata il
         12 febbraio 2010</p>

      <!-- Mini linguaggio con funzione in
           preparsing nella condition -->
      <h3>Modifica del paziente</h3>
      <!-- Funzione in postparsing -->
      <div class="content">
        <h3>Scheda di modifica</h3>
        <form action="?q=object/detail_edit">
          <!-- ... -->
        </form>
      </div>
    </div>
  </div>
@END_TEMPLATE@
```

Come ultima azione, il Page Composer elimina i delimitatori `@BEGIN_-`

`TEMPLATE@` ed `@END_TEMPLATE@` e ritorna il risultato al Serena Method che l'ha invocato. Il Serena Method solitamente non fa altro che prendere questo risultato, ritornarlo alla servlet `Index`, la quale infine lo passa al client richiedente.

Si conclude così il ciclo di vita tipico di una richiesta HTTP a una applicazione Serena.

Esiste però un altro scenario di utilizzo del Serena AS che va approfondito: l'interazione di un'applicazione Serena con altre applicazioni all'interno di una rete. A ciò è dedicata la seguente sezione.

3.9 Come comunica un'applicazione Serena con altre applicazioni

Un'applicazione Serena è in grado di comunicare non solo tramite interazione con un utente ma anche con altre applicazioni. Tale comunicazione può avvenire usando come protocollo XSerena, nel caso in cui anche le altre applicazioni siano applicazioni Serena, oppure usando SOAP tramite Web Service.

Nel caso in cui la comunicazione avvenga tramite XSerena, l'applicazione Serena espone all'esterno i cosiddetti Serena Node, che eventualmente possono essere raggruppati e interrogati contemporaneamente attraverso un Serena Gateway. Più Serena Node raggruppati formano un Serena Virtual Network. Nel caso in cui la comunicazione avvenga tramite SOAP, l'applicazione Serena espone dei webservice creati attraverso la libreria Serena Axis.

Le tre soluzioni (singolo Serena Node, Serena Node raggruppati e webservice) verranno esposte nelle sezioni seguenti.

3.9.1 Serena Node

Un Serena Node è un'interfaccia verso l'esterno di un Serena Persistence[10]. Va interrogato tramite una chiamata HTTP di tipo POST passandogli un parametro con nome `xml` contenente la richiesta XSerena da passare al Serena Persistence.

Non essendoci una fase preliminare di *login*, ogni richiesta viene riconosciuta come effettuata da un utente unico denominato `RemoteUser`. Dall'applicazione Serena è possibile impostare i suoi permessi. Qualora fosse necessaria una maggiore autenticazione, si può procedere attraverso l'uso di scambio di certificati[11]. La gestione multiutente dei nodi è in programma tra gli sviluppi futuri del sistema.

Ogni applicazione Serena ha la configurazione dei propri nodi all'interno della directory `app/conf/system` nel file `config_node.xml`. In esso sono contenute per ogni node alcune informazioni, in particolare un suo *URI* e l'eventuale elenco dei Serena Virtual Network di cui fa parte.

All'interno del Serena AS esistono i seguenti nodi che si possono esporre verso l'esterno.

- `PersistenceAuthNode`. Nodo che filtra la richiestra tramite Serena Auth e la invia al Serena Persistence. È consigliato per i nodi a cui un client accede in modo diretto.

- `Persistence2GatewayNode`. Oltre alle caratteristiche del nodo precedente, gestisce le conversioni dalla propria ontologia a quella condivisa tra tutti i nodi del Serena Virtual Network. Maggiori dettagli alla sezione successiva.

Nel caso si abbiano richieste più particolari, è comunque possibile sviluppare il proprio Serena Node, semplicemente implementando una classe Java figlia di una di quelle che rappresenta un Serena Node esistente[12].

[10] Si veda la sezione 3.6.

[11] Per un approfondimento sulla sicurezza si veda 4.2.

[12] Per approfondimenti si rimanda alla documentazione ufficiale di Serena [Coo10].

3.9.2 Serena Virtual Network e Serena Gateway

I Serena Node facenti parte dello stesso Serena Virtual Network devono avere un frammento di ontologia condivisa per poter dialogare tra loro. Ogni Serena Node di un Serena Virtual Network avrà tra i suoi file di configurazione all'interno della directory `app/conf/system` due file XSLT[13] per ogni Serena Virtual Network di cui fa parte: `<nomesvn>_bridge_in.xsl` e `<nomesvn>_bridge_out.xsl`, rispettivamente per la conversione da e per il resto del Netowrk (a `<nomesvn>` va sostituito il nome del Serena Virtual Network a cui ci si riferisce).

L'interrogazione dei nodi di un Serena Virtual Network avviene tramite una richiesta HTTP POST (con parametro `xml` contenente la richiesta) ad un nodo particolare chiamato Serena Gateway.

Un Serena Gateway si occupa di identificare i nodi a cui è destinata la richiesta, inviarla a ognuno di loro e mettere insieme le risposte ricevute.

La struttura generica di una richiesta a un Serena Gateway è la seguente:

```
<serena action="request">
  <destination
     <Node>
        <condition>
          <uri>{URI del nodo di destinazione}</uri>
        </condition>
      </Node>
    </nodes>
  </destination>
  <service name="{Nome del servizio richiesto}">
    {Parametri del servizio richiesto}
  </service>
</serena>
```

[13] XSLT è un dialetto XML per la trasformazione di documenti da un formato a un altro. Per maggiori informazioni si veda [Kay00].

Come si può notare la richiesta è molto simile a quelle interpretabili da Serena Persistence[14]. L'unica peculiarità è l'esistenza di un tag `destination` contenente informazioni per identificare i nodi destinatari.

Il tag `destination` contiene una serie di tag `Node`, dentro i quali in un tag `condition` è possibile inserire informazioni per identificare uno o più nodi. I nodi possono essere identificati tramite `URI`, `thematic_area` (ambito lavorativo) o `territorial_area` (area geografica): ogni nodo, infatti, nei tipici scenari del Serena Virtual Network, fa riferimento a una posizione geografica (esempi: Porretta Terme, Bologna oppure Ospedale Bellaria, CED, ...) e/o a un ambito di lavoro (esempi: diagnosi, *screening*, oppure neuropsichiatria, neurologia, ...). Tramite le informazioni identificative di un nodo è possibile dunque interrogare più nodi contemporaneamente, raggruppandoli per posizione geografica e/o per area tematica: ad esempio si possono interrogare contemporaneamente tutti i nodi di un Serena Virtual Network collocati a Bologna, oppure tutti i nodi di un Serena Virtual Network che sono relativi a dati per la diagnosi. Tutte le informazioni identificative di un nodo sono reperibili all'interno del file `config_node.xml` delle applicazioni Serena che fanno uso di Serena Node.

Qualora il tag `destination` fosse assente o vuoto, la richiesta verrà inviata a tutti i nodi del Serena Virtual Network di cui il Serena Gateway fa parte.

Il tag `service` è uguale a quello di Serena Persistence, oltre al fatto che supporta anche richieste di tipo `ping`, ovvero richieste vuote che servono solo per controllare che un nodo sia ancora vivo.

Un esempio di richiesta inviata a un Serena Gateway è il seguente:

```
<serena>
 <profile>admin</profile>
  <destination>
    <Node>
      <condition>
```

[14] Si veda la sezione 3.6.

```
        <uri>svn://nodo1</uri>
      </condition>
    </Node>
    <Node>
      <condition>
        <uri>svn://nodo2</uri>
      </condition>
    </Node>
    <Node>
      <condition>
        <uri>svn://nodo3</uri>
      </condition>
    </Node>
  </destination>
  <service name="get">
    <Paziente operation="select" target="?">
      <condition>
        <cognome>Rossi</cognome>
      </condition>
      <cognome />
      <nome>
    </Paziente>
  </service>
</serena>
```

Con questa richiesta verrà inviato ai nodi `nodo1`, `nodo2` e `nodo3` una richiesta di tipo *get* per avere il nome e il cognome di tutti i pazienti con cognome Rossi.

Una volta ricevuta una richiesta, il Serena Gateway passa ai nodi solo la parte relativa al tag `service`. Un esempio di risposta di Serena Gateway alla richiesta precedente è il seguente:

```
<serena action="response">
```

```
<service name="get">
  <result dimension="2" succesful_nodes="2">
    1
  </result>
  <Paziente>
    <metadata>
      <sources>
        <source>
          <uri>svn://nodo2</uri>
          <description>ASL 1</description>
        </source>
      </sources>
    </metadata>
    <nome>Mario</nome>
    <cognome>Rossi</cognome>
  </Paziente>
  <Paziente>
    <metadata>
      <sources>
        <source>
          <uri>nodo2</uri>
          <description>ASL 2</description>
        </source>
      </sources>
    </metadata>
    <nome>Giovanni</nome>
    <cognome>Rossi</cognome>
  </Paziente>
</service>
<metadata>
  <errors>
```

```
        <Error>
          <source>
                <Node>
                  <uri>svn://nodo3</uri>
                </Node>
          </source>
          <code>400</code>
          <message><![CDATA[TIMEOUT]]></message>
        </Error>
      </errors>
    </metadata>
</serena>
```

In questa risposta di esempio il `nodo1` e il `nodo2` hanno risposto correttamente con i dati in loro possesso (un paziente ciascuno). Il `nodo3` è andato invece in *timeout*.

3.9.3 Serena Axis

Si è visto come far comunicare un'applicazione Serena con altre applicazioni Serena o comunque con applicazioni in grado di dialogare usando il protocollo XSerena. Negli scenari d'uso reali però quasi sempre si ha a che fare con applicazioni di parti terze che vogliono usare protocolli standard come ad esempio SOAP.

Esiste perciò Serena Axis, un componente di Serena AS, che, facendo uso della più famosa libreria Apache Axis[15], facilita la creazione di webservice.

Concretamente è possibile creare in ogni applicazione Serena una classe Java figlia della classe astratta `SerenaWebService`: attraverso opportuna configurazione del sistema[16] le funzioni pubbliche di tale classe saranno con-

[15] Axis è una libreria della Apache Foundation per semplificare la creazione e il ciclo di vita dei webservice. Per maggiori informazioni si veda [IB02].

[16] Per approfondimenti si veda [Coo10].

vertite in funzioni remote contattabili da applicazioni esterne. Automaticamente verrà creato il relativo WSDL.

Si conclude così tutta la panoramica della struttura del Serena AS. Si è visto come esso dia la possibilità a un'applicazione Serena di comunicare con un browser o con un'altra applicazione esterna.

La chiamata tramite browser segue solitamente il flusso seguente: chiamata alla servlet `Index`, identificazione del Meta-Ambiente corrente, identificazione dei Serena Module da chiamare, chiamata ai Serena Module, interrogazione del database attraverso Serena Auth e Serena Persistence, eventuale manipolazione dei dati, renderizzazione grafica attraverso Serena Presentation e infine presentazione all'utente.

La chiamata tramite altra applicazione può avvenire interrogando con protocollo XSerena un Serena Node (che di fatto è un'interfaccia al database), a un gruppo di Serena Node contattabili interrogando Serena Gateway oppure con protocollo SOAP interrogando uno o più webservice creati facendo uso di Serena Axis.

Dopo aver analizzato la struttura di tutto il Serena AS, verrà effettuata nel capitolo seguente un'analisi della sua robustezza.

Capitolo 4

ROBUSTEZZA E CONFORMITÀ

Dopo aver analizzato nel dettaglio il Framework Serena e il Serena Application Server, in questo capitolo verrà effettuata un'analisi della robustezza di tutto il sistema Serena e la sua conformità agli obblighi di legge.

In particolare verrà effettuata un'analisi sugli strumenti utilizzati per effettuare i test del software, sulla sicurezza del sistema (e sulle sue potenziali falle) e su come le applicazioni Serena rispettano gli obblighi di legge circa l'accessibilità e la privacy (considerando che uno dei target principali delle applicazioni Serena sono le Pubbliche Amministrazioni e la Sanità).

4.1 Test

Periodicamente il Serena Framework e il Serena Application Server vengono sottoposti a sessioni di test programmati. I test vengono effettuati con più modalità e a più livelli.

I componenti più piccoli e controllabili vengono testati attraverso *JUnit*[1], un sistema di test completamente automatizzato. I test tramite JUnit consistono nel creare tramite il linguaggio Java gli scenari d'uso dei singoli

[1]Per approfondimenti su JUnit si veda [Bec04].

componenti, provando a chiamarli con tutti gli input possibili e indicando a JUnit quali sono gli output che ci si aspetta. Con questa modalità vengono testati:

- Il Serena Persistence[2], provando tutti i tipi di chiamate su una modellazione ontologica di prova (con all'interno tutti i tipi di dato possibili).
- Il Serena Auth[3], provando tutti i tipi di chiamate su una modellazione ontologica di prova (con all'interno tutti i tipi di dato possibili).
- Il Function Parser, chiamando tutte le Serena Function, sia in *preparsing* sia in *postparsing*[4].
- L'Option Parser, utilizzando tutti i costrutti del mini-linguaggio Serena[5].
- Un Serena Node[6], provando tutti i tipi di chiamate su una modellazione ontologica di prova (con all'interno tutti i tipi di dato possibili).
- Un Serena Gateway[7], provando tutti i tipi di chiamate su una Serena Virtual Network con tutti i Serena Node possibili.

Chiaramente le parti più complesse del sistema non sono gestibili con sistemi di test come JUnit. Anche gli stessi Persistence, Auth, Node e Gateway hanno una varietà di casi d'uso talmente vasta da non poter essere considerati robusti solo perché hanno superato la casistica di test di JUnit.

I test di più ampio respiro vengono affidati a un team di *beta tester*: vengono pianificati attraverso il software open source *Testopia*[8] e vengono

[2]Per un approfondimento sul Serena Persistence si veda la sezione 3.6.

[3]Per un approfondimento sul Serena Auth si veda la sezione 3.7.

[4]Per un approfondimento sulle Serena Function e sul Function Parser si veda rispettivamente le sezioni 2.7.4 e 3.8.

[5]Per un approfondimento sul mini-linguaggio Serena e sull'Option Parser si veda rispettivamente le sezioni 2.7.4 e 3.8.

[6]Per un approfondimento sui Serena Node si veda la sezione 3.9.1.

[7]Per un approfondimento sui Serena Gateway si veda la sezione 3.9.

[8]Per approfondimenti su Testopia si veda [Hen10].

eseguiti in parte manualmente dai *beta tester* e in parte in automatico, attraverso *Test Complete*[9], un software che registra le azioni effettuate all'interno di un'applicazione (tasti premuti, movimenti e click del mouse ecc.) e li ripete in automatico, analizzando che il risultato delle azioni eseguite sia quello atteso.

Infine, prima del rilascio di ogni nuova versione di Serena, viene effettuata dai *beta tester* anche una sessione di "test libero": l'utilizzo del Serena Framework e di alcune applicazioni Serena create ad hoc in modalità casuale per cercare di rilevare eventuali bug non riscontrabili dai test programmati.

Pur non essendo direttamente riconducibile alla robustezza del software, una componente importante di un software come Serena è la sicurezza che riesce a garantire, in termini di controllo sulle operazioni ammesse per ogni utente. Questo aspetto viene analizzato nella sezione seguente.

4.2 Sicurezza

Come si può vedere nelle sezioni 2.4 e 3.7, le applicazioni Serena permettono un alto grado di personalizzazione dei permessi di accesso di ogni utente. Il rispetto di tali permessi è garantito dal componente Serena Auth, che si occupa di filtrare le richieste al database in base ai permessi dell'utente correntemente in sessione.

Si noti che l'utilizzo di Serena Auth non è obbligatorio: ad esempio non è utilizzato dal Serena Module `Login`, in quanto deve avere accesso ai dati relativi alle credenziali di accesso, pur non essendoci ancora nessun utente *loggato*.

La sicurezza è comunque garantita dal fatto che non c'è modo di accedere direttamente al database se non passando da un Serena Module o da un Serena Node: dunque l'unica possibilità di accesso malevolo ai dati è attraverso l'installazione di un Serena Module e/o di un Serena Node male-

[9]Per approfondimenti su Test Complete si veda [Tad09].

volo all'interno del server fisico che ospita l'applicazione Serena e il relativo database.

Come visto nella sezione 3.9.1, ogni richiesta a un Serena Node viene riconosciuta come effettuata da un utente unico denominato `RemoteUser`. Ciò rappresenta una delle più gravi mancanze del sistema Serena, in quanto non è possibile garantire una migliore profilazione degli accessi ai Serena Node. Tra gli sviluppi futuri di Serena c'è sicuramente la risoluzione di tale mancanza. Al momento la si può aggirare creando un Serena Node per ogni profilo e gestendo le autorizzazioni di accesso attraverso certificati SSL, gestibili a monte dal *servlet container*.

Oltre a robustezza e sicurezza, vengono fatte per Serena ulteriori analisi per riscontrare la compatibilità con i dettami relativi all'accessibilità e al rispetto della privacy, come mostrato nelle sezioni seguenti.

4.3 Accessibilità

Fin dalla sua nascita, Serena ha tra i suoi obiettivi la realizzazione di applicazioni web accessibili, cioé fruibili da tutti (siano normodotati, disabili visivi, audiolesi, disabili motori ecc.), secondo il concetto tecnico ed etico del *design for all*[10], sia per uno dei potenziali *target* dell'applicazione (le Pubbliche Amministrazioni e la Sanità in generale), sia per la *mission* della Cooperativa Anastasis.

I riferimenti legislativi e tecnologici per tale obiettivo sono la legge n. 4 del 9 gennaio 2004 (la cosiddetta "Legge Stanca"[11]) e le WCAG 2.0 del W3C[12].

Le caratteristiche che fanno di Serena un framework di sviluppo di applicazioni web accessibili sono le seguenti:

[10]Si veda [KRH+06].
[11]Si veda [Sca05].
[12]Si veda [W3C01].

- L'interfaccia grafica predefinita del sistema (il cosiddetto Meta-Ambiente, analizzato nella sezione 2.7.1 e i Template e i Meta-Template, descritti nella sezione 2.7.2) è fruibile anche attraverso browser testuale o *screen-reader* e navigabile anche senza l'ausilio del mouse. Ad esempio tutti i contenuti grafici hanno un alternativa testuale, le sezioni principali dell'applicazione sono raggiungibili attraverso tasti di accesso rapido, il *layout* delle pagine è fluido con testo ingrandibile. L'HTML risultante è XHTML standard e supera i controlli principali dei validatori del W3C e di alcuni validatori esterni (come *Cynthia*).

- In ogni applicazione Serena funzionante, come visto nella sezione 2.1.3, le parti di testo esteso inseribili dall'utente, che potenzialmente potrebbero pregiudicare l'accessibilità delle pagine risultanti, vengono inserite attraverso il *PegoEditor*, un editor HTML WYSIWYG sviluppato dalla Cooperativa Anastasis con lo specifico fine di creare HTML accessibile[13].

Al momento è comunque possibile realizzare Meta-Ambienti o (Meta) Template non accessibili. Tra gli sviluppi futuri del sistema (si veda il capitolo 6) si può pensare a un validatore di Meta-Ambienti e (Meta) Template.

4.4 Privacy

Poiché spesso il Framework Serena è utilizzato per creare cartelle cliniche e in generale applicazioni in ambito sanitario, è stato pensato e progettato per rispettare le direttive legislative nazionali sul rispetto della privacy. Il riferimento principale è il "Disciplinare tecnico in materia di misure minime di sicurezza", allegato B della Legge 196[14].

In particolare la conformità alla legge è garantita da:

[13]Per approfondimenti sul PegoEditor si veda [Ana10].
[14]Si veda [Gia97].

- La possibilità, attraverso opportuna configurazione del *servlet container*, di effettuare le comunicazioni in modo criptato utilizzando il protocollo HTTPS.

- Un Serena Module specifico (`userRegistration`, non esposto in questa tesi) si occupa di una gestione raffinata degli utenti, imponendo ad ogni utente di avere una password di almeno otto caratteri e di cambiarla ogni tre mesi.

- Per gli slot relativi a dati anagrafici identificativi (ad esempio nome, cognome e codice fiscale dei pazienti) e dati relativi allo stato di salute e la vita sessuale è possibile utilizzare come tipo di dato la "stringa criptata"[15]. I tipi di dato del genere vengono memorizzati nel database in modo criptato attraverso l'algoritmo di cifratura DES, con chiave salvata su *filesystem* (e non leggibile da utenti di sistema diversi da quello che avvia l'*application server*). La cifratura fa sì che anche chi ha accesso diretto al database non può mettere in relazione i dati sensibili mostrati in chiaro con i dati anagrafici delle persone a cui si riferiscono. A maggior sicurezza su questo aspetto, il team di sviluppo sta valutando per gli sviluppi futuri l'utilizzo di algoritmi di cifratura ancor più robusti di DES.

- La password di accesso di ogni utente è essa stessa una 'stringa criptata" affinché siano offuscate anche le credenziali di accesso del personale sanitario.

- È possibile impostare i permessi di accesso ai dati in modo molto granulare su oggetti e istanze in base a gruppi di utenti, come visto nella sezione 2.4.

- Il Serena Application Server è stato progettato tenendo presenti i principali tipi di attacchi web (*SQL injection*, *Cross-site scripting*, *Command injection* ecc.).

[15] Sulla configurazione dei dati si vedano le sezioni 2.1.2, 2.1.3 e 2.6.

Si conclude così l'analisi della robustezza di Serena, cioé quali meccanismi sono stati utilizzati dal team di sviluppo per testare il sistema e per garantire il rispetto dei dettami su accessibilità e privacy.

Nel prossimo capitolo si tornerà ad analizzare il Serena Framework e il Serena Application Server, sottolineando, alla luce di quanto visto nei capitoli precedenti, i pregi ed i difetti del sistema.

Capitolo 5

PREGI E DIFETTI DI SERENA

Nel capitolo 1 è stato mostrato lo scenario attuale in termini di prodotti esistenti riguardo i framework di sviluppo di applicazioni web, i framework di applicazioni basate su ontologie e gli application server. In questo capitolo si tornerà sull'argomento, avendo nel frattempo chiarito struttura e funzionamento del Framework Serena e del Serena Application Server (rispettivamente nei capitoli 2 e 3), per mostrare pregi e difetti del sistema Serena rispetto ai software similari.

La prima cosa che salta agli occhi è che Serena colma un vuoto: esistono valide soluzioni in termini di framework di sviluppo di applicazioni web, di framework di sviluppo di ontologie e di application server, ma nessuna soluzione stabile che integri insieme i tre concetti.

Inoltre, essendo basato su ontologie, Serena ha il valore aggiunto (spesso mancante nelle altre soluzioni) di poter utilizzare un sistema esperto: tale caratteristica esiste fin dalla fase progettuale di Serena ma finalmente dalla versione 1.5 è integrata e testata, grazie al Serena Module `Expert System` descritto nella sezione 2.5.

L'unica soluzione pre-esistente, almeno in ambito open source, che comprende un framework di sviluppo, un application server e un sistema esperto

è JBoss (che usa rispettivamente JBoss Seam come framework, JBoss Rules come sistema esperto ed è esso stesso un application server).

Naturalmente non si ha la pretesa di mettere sullo stesso piano Serena e JBoss, sia in termini di robustezza, sia di numero di organizzazioni e persone coinvolte, sia di diffusione sia di organicità e pulizia progettuale. Se però, pur esistendo qualcosa come JBoss, si è comunque deciso di creare un sistema ex novo, è perché si voleva avere un qualcosa che, pur al prezzo di limitare le funzionalità e gli ambiti di intervento, avesse una curva di apprendimento bassa e un'alta velocità di sviluppo (soprattutto per la creazione di un primo prototipo).

Le prove sul campo sembrano dare conforto all'ipotesi che tali obiettivi siano stati raggiunti:

- la Cooperativa Anastasis in passato ha svolto corsi di formazione per lo sviluppo di applicazioni tramite Serena. Si è così visto che è possibile formare nuovi sviluppatori (con conoscenze di base di Java, HTML, CSS e teoria di sistemi middleware) in soli due giorni di lezione teoriche e un giorno di lezioni pratiche.

- la Cooperativa Anastasis mediamente riesce a creare, a partire da una prima intervista con il cliente, un primo prototipo funzionante di applicazione utilizzando all'incirca quattro giorni/uomo.

La minor espressività rispetto ad alternative come JBoss è compensata dalla scalabilità resa possibile dai Serena Module: ogni sviluppatore anche esterno alla Cooperativa Anastasis può modificare i comportamenti del sistema semplicemente creando nuovi moduli, purché rispettino gli standard descritti nella sezione 3.5.

Appurato che Serena ha, per quanto appena visto, una sua ragion d'essere, si può affermare anche che il suo più grande difetto è, sui singoli componenti, aver in diversi ambiti "reinventato la ruota" quando invece sarebbe stato più veloce e robusto appoggiarsi su realtà preesistenti.

Nelle prossime sezioni verrà fatto a tal proposito un breve confronto tra ciò che in Serena è stato reinventato e le soluzioni esistenti più famose.

5.1 Modello ontologico vs modello relazionale

Come visto nella sezione 2.1.2, serena usa il modello ontologico a frame gestito dal software Protégé. Si è scelto di usare in Serena il modello ontologico, preferendolo al classico modello relazionale, principalmente perché ciò rendeva possibile l'uso di un sistema esperto, ma ciò non esaurisce le differenze tra i due modelli.

Un'ontologia Serena, come visto nella figura 2.1, è un grafo aciclico: conseguentemente, fissando un nodo come radice, è immediatamente convertibile in un albero e quindi in un documento XML (Protégé stesso fornisce delle API per farlo).

Sia le ontologie Protegé che quelle Serena hanno inoltre alcuni vantaggi minori, non presenti nel modello relazionale:

- Lo stesso slot (come ad esempio uno slot `Nome` di tipo `String`) è utilizzabile da più classi, riducendo il tempo di modellazione e le possibilità di errori di tipizzazione.

- Esistono gli slot di tipo `Symbol`, che vincolano i contenuti a un *range* ristretto di dati.

Le ontologie Protégé hanno anche alcuni vantaggi in più, soprattutto relativamente ai vincoli sui dati (ad esempio è possibile indicare un limite minimo e massimo nella cardinalità delle relazioni), che rendono il modello ontologico più espressivo rispetto a quello relazionale. Tali caratteristiche non sono però ancora supportate da Serena: tale supporto è tra gli sviluppi futuri del sistema.

5.2 XSerena vs SOAP

Il protocollo XSerena poteva essere sostituito dal più noto protocollo SOAP. Si è preferito non farlo in quanto messi a confronto i due protocolli, XSerena è più snello e più facile da utilizzare.

Ecco ad esempio come viene descritta l'ontologia della figura 2.1 in XSerena

```
<serena action="response">
  <service name="get">
    <Paziente>
      <codice_fiscale>CRSGNS02A41A944Q</codice_fiscale>
      <medico_di_famiglia>
        <Medico>
          <nome>Marco</nome>
          <cognome>Lelli</cognome>
        </Medico>
      </medico_di_famiglia>
    </Paziente>
  </service>
</serena>
```

Ed ecco come viene descritta la stessa ontologia con SOAP:

```
<soap:Envelope
  xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
  soap:encodingStyle=
       "http://www.w3.org/2001/12/soap-encoding">
  <soap:Header>
    <m:Trans
       xmlns:m="http://www.w3schools.com/transaction/"
       soap:mustUnderstand="1">234
    </m:Trans>
  </soap:Header>
```

```
<soap:Body xmlns:m="http://www.example.org/stock">
  <m:GetService>
    <m:GetServiceReturn>
      <p:Paziente
          xmlns:p=
          "http://serena.anastasis.it/Paziente">
        <p:codice_fiscale>
          CRSGNS02A41A944Q
        </p:codice_fiscale>
        <p:medico_di_famiglia>
          <m:Medico
              xmlns:p=
              "http://serena.anastasis.it/Medico">
            <m:nome>Marco</m:nome>
            <m:cognome>Lelli</m:cognome>
          </m:Medico>
        </p:medico_di_famiglia>
      </p:Paziente>
    </m:GetServiceReturn>
  </m:GetService>
</soap:Body>
</env:Envelope>
```

Si sarebbe però potuto sopperire alla complessità di SOAP attraverso serializzatori e deserializzatori affinché la comunicazione tra componenti e con l'esterno fosse stata trasparente al programmatore, che non avrebbe mai lavorato direttamente con l'XML bensì direttamente con oggetti *wrapper*. Tali strumenti esistono ormai per tutti i linguaggi di programmazione più famosi: per Java ad esempio esiste Jibx[1].

Bisogna dire che comunque Serena è in grado di comunicare con applicazioni esterne usando SOAP, attraverso il componente Serena Axis.

[1]Per approfondimenti si veda [TVM+03].

5.3 Serena Persistence vs Hibernate

La gestione della persistenza dei dati di Serena, come visto nei capitoli precedenti, è affidato al Serena Persistence. Il compito del Serena Persistence è praticamente identico a quello di componenti più famosi come Hibernate[2].

L'uso di Hibernate avrebbe sicuramente garantito maggior stabilità e minori giorni di sviluppo. Hibernate non avrebbe però nativamente avuto il supporto a XSerena, cosa che, come appena visto, si sarebbe potuta ovviare tramite serializzatori e deserializzatori.

Il passaggio da Serena Persistence a Hibernate è previsto tra gli sviluppi futuri.

5.4 Minilinguaggio vs JSP

Essendo ospitato da un *servlet server*, Serena avrebbe potuto supportare il linguaggio JSP a costo zero. Si è preferito scartare questa ipotesi perché JSP non garantisce una valida separazione tra logica e presentazione di un'applicazione.

A volte però si è comunque reso necessario l'uso di piccoli costrutti di controllo (ad esempio dei semplici *if-then-else*) nei Serena Template e si è dovuto quindi costruire un mini-linguaggio.

Ecco ad esempio un costrutto *if-then-else* in un Serena Template:

```
@BEGIN_TEMPLATE@
  @BEGIN_Paziente@
    ([
      [[@tag_nome#FLAT@] == [Mario]]
      ?? [\'E il solito paziente Mario.]
      :: [Finalmente qualcuno che non si chiama Mario!]
    ])
  @END_Paziente@
```

[2]Per maggiori informazioni su Hibernate si veda [BK06].

```
@END_TEMPLATE@
```

Ed ecco come sarebbe stato se i Serena Template supportassero JSP:

```
@BEGIN_TEMPLATE@
  @BEGIN_Paziente@
    <%!
      nome = @tag_nome#FLAT@

      if (nome == "Mario)
      {
        %>\'E il solito paziente Mario.<%!
      } else
      {
        %>Finalmente qualcuno che non si chiama Mario!<%!
      }
    %>
  @END_Paziente@
@END_TEMPLATE@
```

Come già visto nella sezione 2.7.4, il mini-linguaggio Serena supporta solo i costrutti *if-then-else*, mentre JSP ha di fatto la stessa espressività del linguaggio di programmazione Java.

Tale scelta progettuale limita fortemente l'espressività dei Serena Template: in particolare non esistono costrutti per gestire cicli e gli unici cicli possibili sono quelli sui dati durante la navigazione ontologica (come ad esempio nelle liste di oggetti). È però una scelta voluta, per "costringere" gli sviluppatori a spostare ogni parte logica fuori dai template, attraverso l'uso di Serena Module specifici e/o attraverso le Serena Function. Di conseguenza ogni parte logica delle applicazioni Serena passa necessariamente da parti scritte in Java, nel pieno rispetto del pattern Model View Controller.

5.5 Serena Template vs XSLT

Per ottenere la separazione tra parte logica e parte presentazionale (scelta analizzata nella sezione precedente), si sarebbe comunque potuto adottare per i Serena Template il linguaggio XSLT. Si è scelto di preferire ad XSLT una sintassi propria per i seguenti motivi:

- XSLT è estremamente "verboso" e usarlo per scrivere Serena Template complessi ne avrebbe compromesso estremamente la leggibilità e avrebbe reso complesso seguire il flusso di costruzione delle pagine finali.
- L'esecuzione dei Serena Template attuali, proprio a causa della loro minore espressività, è estremamente più veloce dell'esecuzione di un equivalente foglio XSLT (in base ad alcuni test effettuati dal team di sviluppo).
- Il formato usato dai Serena Template era già usato all'interno della Cooperativa Anastasis per un prodotto precedente (chiamato *CMS Accessibile*) e manterlo avrebbe permesso un basso *switch cost* per il personale che si occupa della grafica all'interno dell'azienda.

Si può comunque pensare tra gli sviluppi futuri di Serena un supporto a Serena Template in XSLT, eventualmente con un convertitore da un formato all'altro.

5.6 Serena Template vs Hamlets

Un discorso analogo a quello fatto nella sezione precedente su XSLT, può essere fatto per lo standard JSTL e/o per Hamlets[3], facilitando l'apprendimento a realtà esterne all'azienda.

[3]Hamlets è un estensione delle servlet nato per separare la parte logica da quella grafica. Fondamentalmente è composto da template HTML a cui sono aggiunti i tag `repeat` e `replace`, automaticamente gestiti dopo tramite codice Java. Per approfondimenti si veda [Paw10].

Ecco ad esempio com'è un Serena Template;

```
@BEGIN_TEMPLATE@
  <div class="title">
    <h3>Scheda di dettaglio</h3>
  </div>
  <div class="content">
    @BEGIN_Paziente@
      @tag_cognome@
      @tag_nome@
    @END_Paziente@
  </div>
@END_TEMPLATE@
```

Ed ecco come sarebbe potuto essere utilizzando Hamlet:

```
<repeat id="TEMPLATE">
  <div class="title">
    <h3>Scheda di dettaglio</h3>
  </div>
  <div class="content">
    <repeat id="Paziente">
      <replace id="cognome" />
      <replace id="nome" />
    </repeat>
  </div>
</repeat>
```

In particolare Hamlets (al contrario di XSLT) passa immediatamente il controllo delle iterazioni e delle sostituzioni a Java, minimizzando i problemi di performance visti con XSLT pur mantenendo un'ottima separazione tra logica e presentazione.

Il supporto ad Hamlets, però, stravolgerebbe talmente la gestione attuale della parte presentazionale da non essere auspicabile nel breve termine, a

causa di un eccessivo *switch cost*.

L'analisi effettuata in questo capitolo, relativa agli aspetti positivi da incentivare e a quelli negativi da eliminare, verrà presa in considerazione dalla Cooperativa Anastasis e inciderà sugli sviluppi futuri dei componenti di base del sistema Serena. Contemporaneamente, a livello più macroscopico, la Cooperativa Anastasis sta già mettendo in atto nuovi progetti, che accompagnano e approfondiscano la via tracciata con il sistema Serena.

Tutti gli sviluppi futuri, sia relativi ai componenti di base del sistema Serena che a progetti di contorno verranno analizzati meglio nel capitolo seguente.

Capitolo 6

SVILUPPI FUTURI

Serena, pur avendo raggiunto una stabilità tale da permettere lo sviluppo di applicazioni importanti come le cartelle cliniche sanitarie, è sempre in continua evoluzione: alcuni sviluppi futuri minori verranno esposti nella sezione 6.1.

Il team di sviluppo è impegnato inoltre su due linee evolutive: la semplificazione dell'utilizzo di Serena Framework e lo sviluppo di versioni di Serena specializzate ad ambiti vicini al *core business* della Cooperativa Anastasis.

La prima linea di sviluppo si va concretizzando attraverso la creazione di un plugin per Eclipse che automatizza gran parte della configurazione e personalizzazione delle applicazioni Serena. La seconda linea si va concretizzando con lo sviluppo di un nuovo framework, chiamato *Bubbles*. Le due linee di sviluppo verranno analizzate rispettivamente nelle sezioni 6.2 e 6.3.

6.1 Sviluppi futuri minori

Nel capitolo 5 sono state evidenziate alcune mancanze del sistema Serena. Tale analisi verrà presa in considerazione dal team di sviluppo per migliorare ulteriormente Serena nella futura versione 1.6. In particolare:

- Le ontologie Serena si basano sulle ontologie Protégé ma non ne supportano tutte le caratteristiche (ad esempio non è gestito il vincolo sulla

cardinalità delle relazioni). Colmare questa mancanza è un possibile sviluppo futuro.

- I Serena Template sono scritti secondo un linguaggio specifico e non standard. Tale scelta è voluta e la si vuole mantenere anche in futuro (come visto nella sezione 5.5). Si può però considerare un valido sviluppo futuro la possibilità di esprimere i Serena Template anche in XSLT, eventualmente con un convertitore da un formato all'altro.

- Nella sezione 5.3 si è visto come Hibernate sia più stabile, più noto e più facile da usare rispetto al Serena Persistence. Prima ancora che venisse iniziata la stesura di questa tesi, il passaggio a Hibernate era già stato preso in seria considerazione come sviluppo futuro di Serena.

- Al momento le comunicazioni con i Serena Node sono *stateless* e non è supportata alcuna fase preliminare di *login*. Conseguentemente non è possibile una profilazione degli accessi ai dati attraverso i Serena Node: ogni richiesta viene riconosciuta come effettuata da un utente unico denominato `RemoteUser` ed è possibile gestire esclusivamente i permessi di tale utente. Dunque, un possibile sviluppo futuro è rendere i Serena Node *stateful* (come già è Serena Application) e pensare a una modalità di login.

- Sempre sul fronte sicurezza, è in programma tra gli sviluppi futuri l'utilizzo di algoritmi di cifratura più robusti rispetto a DES, attualmente utilizzato per la criptazione dei dati sensibili all'interno del database (come visto nella sezione 4.4).

6.2 Serena Eclipse Plugin

Come si può facilmente notare, molte delle operazioni necessarie alla creazione di un'applicazione Serena[1] sono facilmente automatizzabili. E si può

[1] I passi per la creazione di un'applicazione Serena sono descritti nella sezione 2.1.

notare anche che molte delle operazioni di personalizzazione di un'applicazione Serena[2], pur non essendo automatizzabili, sono semplificabili attraverso opportuni *wizard* e facilitatori grafici.

Da queste considerazioni nasce l'idea di creare un plugin Eclipse per il Framework Serena. Il plugin comprenderà:

- Un *wizard* per la creazione di una nuova applicazione Serena. Il *wizard* chiederà allo sviluppatore i dati essenziali di configurazione (nome del progetto, dati di accesso al database ecc.) e creerà un progetto Java nell'ambiente di sviluppo Eclipse già opportunamente configurato.

- Un *query editor*, cioé un editor XML simile a quello già esistente in Eclipse, con auto-completamento, vista ad albero, *syntax highlighting* e validazione delle richieste XSerena.

- Un *template creator* dal comportamento molto simile a quello del già esistente Template Generator descritto nella sezione 2.7.3.

- Un *template editor*, cioé un editor HTML simile a quello già esistente in Eclipse, con auto-completamento, *syntax highlighting* e validazione dei template (sia come struttura che come accessibilità dell'HTML risultante).

- Un *wizard* per la configurazione dei Serena Entity Bean, sulla falsariga del Serena Developer Tool descritto nella sezione 2.1.3, magari migliorandone l'usabilità.

- Un *wizard* per la configurazione dei Serena Interface Bean, sulla falsariga del Serena Developer Tool descritto nella sezione 2.1.3, magari migliorandone l'usabilità.

Al momento della stesura di questa tesi il plugin Eclipse per il Framework Serena è in fase iniziale di sviluppo da parte di Vincenzo Carnazzo.

[2]Le personalizzazioni possibili di un'applicazione Serena e i modi di eseguirli sono descritti nelle sezioni 2.5, 2.6 e 2.7.

6.3 Bubbles Framework

Come descritto nell'introduzione, Serena nasce con l'obiettivo principale di creare un'infrastruttura integrata per la gestione dell'intero processo di "presa in carico" di utenti in situazioni di disagio, dallo *screening*, alla rieducazione alla formazione. Con gli anni il sistema Serena si è evoluto ed è diventato un generico framework di sviluppo di applicazioni web.

I tempi sono maturi adesso per creare una sintesi tra ciò per cui il sistema Serena è nato e ciò che poi è diventato. La Cooperativa Anastasis ha deciso perciò di iniziare a sviluppare nuovi software di riabilitazione per bambini con Disturbi Specifici dell'Apprendimento[3] (che da anni caratterizzano il suo *core business*) utilizzando Serena stessa come framework di base. Ciò dà i seguenti vantaggi:

- I software riabilitativi si integreranno con le cartelle cliniche e in generale i sistemi informativi dei centri sanitari.

- I software riabilitativi potranno essere dotati di un sistema esperto, in modo tale che adattino automaticamente i propri parametri in base ai risultati pregressi dell'utente. Ad esempio, un software per esercitare la lettura accompagnato da sintesi vocale imposterà la velocità di lettura della sintesi in base al numero di errori che l'utente ha commesso nella lettura dei brani passati.

- L'intero sistema potrà essere dotato di un sistema esperto, che attiva/disattiva uno o più software riabilitativi in base alla storia clinica dell'utente e alle scelte effettuate dai riabilitatori.

- L'utente potrà inviare i risultati degli esercizi svolti a un server centrale.

[3]Con Disturbi Specifici dell'Apprendimento (DSA) si identificano le difficoltà che una persona incontra nell'apprendere le abilità scolastiche di base: leggere (dislessia), scrivere (disgrafia) e fare di conto (discalculia). Maggiori informazioni su [Sab95].

- Il clinico potrà visionare i risultati dell'utente, creare statistiche, generare report e utilizzare in generale tutti gli strumenti messi a disposizione da ogni applicazione Serena.

- Il clinico potrà inoltre modificare i parametri di configurazione del software del singolo utente, sostituendosi al sistema esperto.

Per tali scopi si è deciso di creare un framework specializzato "figlio di Serena", denominato *Bubbles*, più limitato di Serena ma con in più alcuni automatismi e alcune caratteristiche che facilitano e velocizzano lo sviluppo dei software riabilitativi descritti sopra.

La progettazione e l'implementazione dell'infrastruttura di Bubbles è stata effettuata da Vincenzo Carnazzo e Matteo Tassetti, in collaborazione con Enzo Ferrari, Andrea Pegoretti e Fabrizio Piazza, della Cooperativa Anastasis. La prima versione del nuovo framework è stata realizzata durante la stesura di questa tesi ed è attualmente in fase di *beta testing*). Tale prima versione è comprensiva del primo software riabilitativo chiamato *Reading Trainer*, un software per esercitarsi nella lettura.

Una schermata di *Reading Trainer* è mostrata nella figura 6.1.

Al contrario di Serena, Bubbles è un framework che non verrà reso pubblico e resterà a uso interno della Cooperativa Anastasis.

6.3.1 Ri-Di

Vista dal punto di vista del personale clinico, l'idea vista nella sezione precedente si concretizza in un'applicazione dove si può gestire l'anagrafica degli assistiti, scegliere quali software riabilitativi somministrare, modificarne i parametri di utilizzo e visionare i risultati ottenuti dagli assistiti nell'uso del software. Tale applicazione è stata sviluppata da Vincenzo Carnazzo per la Cooperativa Anastasis: si chiama Ri-Di (acronimo di Riabilitazione a Distanza) ed è una tradizionale applicazione Serena[4].

[4]L'applicazione Ri-Di è già online e, per chi ha le credenziali di accesso, raggiungibile all'URL `https://ridi.sere-na.it`.

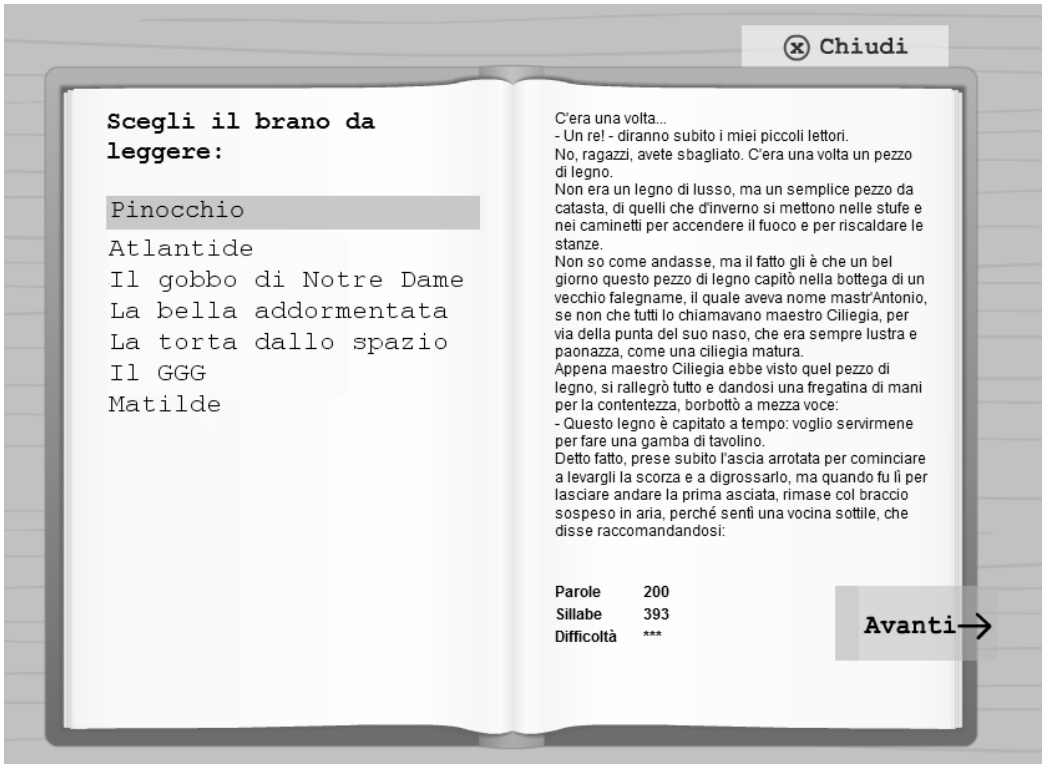

Figura 6.1: Schermata di esempio di Reading Trainer

Al momento Ri-Di gestisce due software riabilitativi pre-esistenti della Cooperativa Anastasis (Tachistoscopio e Sillabe, sviluppati con il fine di far esercitare i bambini nella lettura rispettivamente di parole e di sillabe) e lo scambio tra i due software *legacy* e il server centrale avviene attraverso un client Java aggiuntivo che invia e riceve i dati secondo il protocollo XSerena all'interno di un Serena Virtual Network[5]. In futuro invece i software riabilitativi sviluppati usando Bubbles nasceranno già integrati con Ri-Di.

Trattandosi di una "tradizionale" applicazione Serena, Ri-Di non verrà qui ulteriormente descritta.

Visto invece dal punto di vista di un utente assistito (il bambino che deve fare riabilitazione), il sistema è una console di gioco dove un *trainer* lo segue, indicandogli gli esercizi da eseguire e i risultati ottenuti. Gli esercizi che può eseguire e i relativi parametri di utilizzo sono stabiliti dal personale clinico e/o dal Sistema Esperto in base alla storia clinica dell'assistito. Ogni esercizio è un software riabilitativo diverso.

Si vedrà nella prossima sezione come attraverso il Framework Bubbles è possibile sviluppare nuovi software riabilitativi.

6.3.2 Come creare un nuovo software riabilitativo bubble

Con il framework Bubbles, come accennato nelle sezioni precedenti, è possibile creare software riabilitativi (chiamati "bubble"). I software così creati possono essere eseguiti *stand-alone* (come qualsiasi applicazione client) senza alcuna comunicazione con l'esterno oppure online all'interno di un oggetto Flash e con la possibilità di comunicare con il server Ri-Di. Con il framework Bubbles, lo sviluppatore ha il vantaggio di poter partire da una grande mole di codice auto-generato, con uno scheletro di struttura dati già formato, senza il bisogno di preoccuparsi degli aspetti più specifici della comunicazione

[5]Per approfondimenti sul Serena Virtual Network si veda la sezione 3.9.

con il server centrale e senza la necessità di scrivere codice diverso per le due versioni (*stand-alone* ed online) del software.

Lo sviluppatore dovrà solo concentrarsi soltanto nella scrittura del codice della parte logica del software (per l'avanzamento dei livelli, per la valutazione dei risultati e per il loro salvataggio) in Java e della presentazione grafica del tutto in Flex ed Actionscript[6]. Vedremo ora come.

Come Serena, Bubbles è pensato per applicazioni sviluppate all'interno dell'IDE Eclipse. Inoltre è necessario aver installato Flex Builder[7], Granite DS Eclipse Plugin[8] e Bubbles Eclipse Plugin[9]. Il Bubbles Eclipse Plugin in particolare aggiunge ad Eclipse la possibilità di creare "Progetti Bubbles" e aggiunge un nuovo menu "Bubbles" con alcuni comandi specifici.

Per iniziare, innanzitutto bisogna creare un nuovo Progetto Bubbles (con l'apposito comando in Eclipse): viene così creato un progetto Eclipse con all'interno già tutte le cartelle e i file principali che ogni bubble deve avere:

- `src-java` che conterrà il codice della parte logica in Java
- `src-as3` che conterrà il codice della parte presentazionale in Flex ed Actionscript.
- `protege` che conterrà l'ontologia dei dati
- `webapps` che rispecchia la cartella webapps di Serena[10], con i file di configurazione e i Serena Bean. Non verrà qui analizzata ulteriormente.

[6]Per approfondimenti su Flash, Flex ed Actionscript si veda rispettivamente [DR07], [LK08] ed [Moo01].

[7]Flex Builder è un plugin di Eclipse a pagamento della Adobe, che facilita lo sviluppo di applicazioni in Flex ed Actionscript. Volendo se ne può comunque fare a meno.

[8]Granite DS Eclipse Plugin è un plugin per Eclipse scaricabile gratuitamente da `www.graniteds.org` per creare il codice necessario a Granite DS. Granite DS (Data Services) è la versione open di Blaze DS, l'infrastruttura della Adobe per effettuare chiamate remote da Actionscript e Flex. Per maggiori informazioni si veda [WD10].

[9]Bubbles Eclipse Plugin è un plugin per Eclipse sviluppato dalla Cooperativa Anastasis e fuori commercio.

[10]Si veda la sezione 2.1.1.

Nella radice del progetto è inoltre presente il file `build.xml` con i task ANT necessari alla creazione dell'applicazione: bisogna modificarlo indicando nella variabile `FLEX_HOME` il percorso dove trovare l'SDK Flex.

La cartella `protege` invece conterrà già lo scheletro dell'ontologia della bubble. Tutte le bubble infatti condividono uno scheletro comune che è mostrato nella figura 6.2.

Si noti che:

- `Bubble` è la "radice" dell'ontologia: contiene le informazioni principali (come ad esempio il nome della bubble) e da lì si diramano tutte le relazioni.

- Per `Level` si intende un classico "livello di gioco", cioè un'unità di lavoro con inizio e fine precisa. Può essere ad esempio un brano da leggere.

- Per `Item` si intende un elemento atomico da mostrare all'utente. Può essere ad esempio una parola.

- Ogni `Level` può essere suddiviso in sotto-livelli. Ad esempio un brano può essere suddiviso in capitoli, che a loro volta possono essere suddivisi in paragrafi.

- Ogni Item può essere suddiviso in sotto-*item*. Ad esempio una parola può essere suddivisa in sillabe, che a loro volta possono essere suddivisi in morfemi.

- La parte che in figura è a sinistra contiene i dati relativi alla "struttura" della bubble, cioè quella contenente il nome della bubble, i livelli in cui è suddivisa e gli *item*.

- La parte che in figura è a destra contiene i risultati degli esercizi svolti dall'utente.

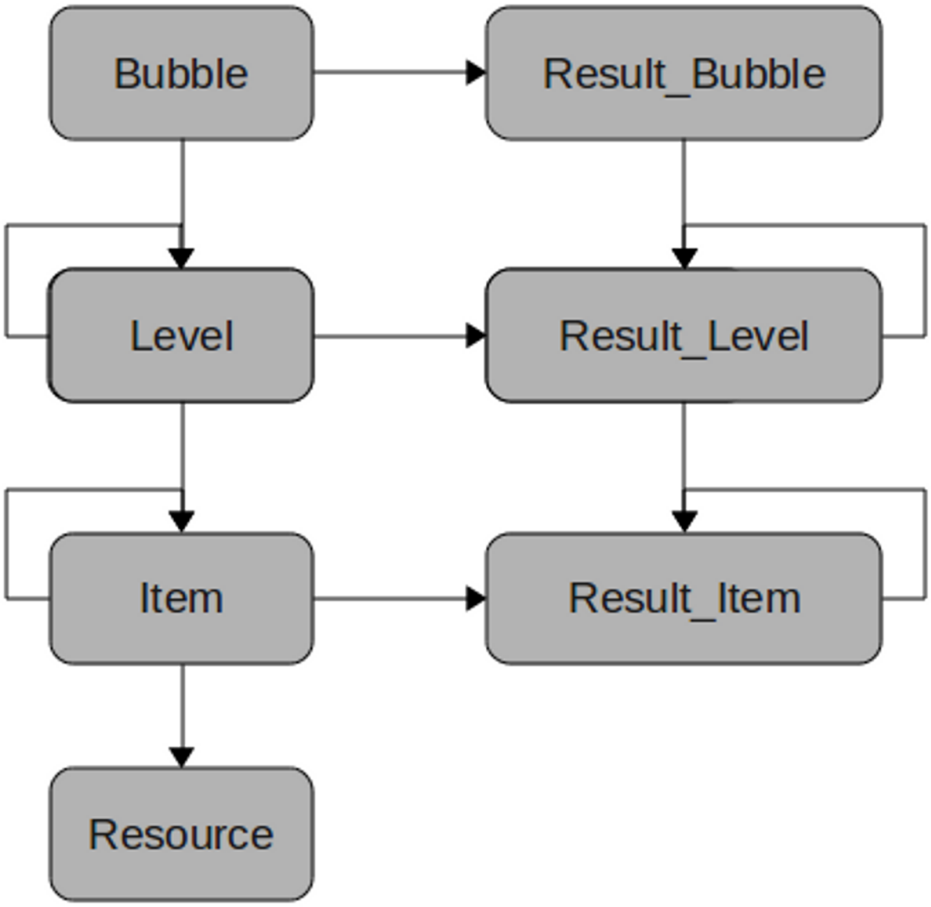

Figura 6.2: Ontologia comune a tutte le bubble

- Nella figura le frecce indicano le relazioni dirette[11]. Si noti che pur essendoci relazioni ricorsive (`Level` e `sub_levels`, Item e `sub_items` ecc.), i dati contenuti in essi sono comunque finiti e quindi non è possibile che gli algoritmi di visita divergano.

A partire dallo scheletro comune di ontologia, il Bubbles Eclipse Plugin, all'atto di creazione del progetto, genera un'ontologia specifica (ad esempio per il software Reading Trainer esistono le classi `ReadingTrainer_Bubble`, `ReadingTrainer_Level`, `ReadingTrainer_Item` ecc.).

Per personalizzare la struttura dati della propria bubble, bisogna dunque aprire con Protégé tale ontologia e modificarla secondo le proprie esigenze. Non è lecito cancellare gli slot esistenti in quanto vengono utilizzati dall'infrastruttura.

Una volta modellata l'ontologia Protegé, i dati possono essere utilizzati attraverso classi *wrapper* in Actionscript (per la parte presentazionale) e in Java (per la parte logica). La creazione di tali classi viene effettuata semplicemente selezionando una voce di menu del Bubbles Eclipse Plugin: il plugin leggerà l'ontologia e genererà automaticamente le classi *wrapper* Java all'interno di `src-java` e, in modo trasparente, Granite DS Eclipse Plugin creerà automaticamente le speculari classi Actionscript all'interno della cartella `src-as3`.

A questo punto si può iniziare a scrivere il codice sorgente vero e proprio della bubble.

Come già detto, la parte presentazionale va sviluppata in Flex ed Actionscript. All'interno della cartella `src-as3` esiste già il file `<nomebubble>.mxml`. È un modulo Flex[12] chiamato *Bubble Module*, che viene attivato automaticamente dal sistema non appena deve partire la bubble che si sta sviluppando. La parte presentazionale va sviluppata concretamente modificando tale modulo Flex.

[11] Per approfondire il concetto di relazioni dirette e inverse si veda la sezione 2.1.3.

[12] I moduli Flex sono oggetti SWF non autonomi: per poter funzionare vanno caricati all'interno di un oggetto SWF contenitore. Per approfondimenti si veda [LK08].

La parte logica va invece sviluppata in Java. All'interno della cartella `src-java` esiste già il file `<nomebubble>Controller.java`: è la classe Java `eu.anastasis.bubbles.<nomebubble>.<NomeBubble>Controller` chiamata *Bubble Controller*. La parte logica va sviluppata concretamente modificando tale classe Java.

Nello sviluppo concreto va considerata la bubble come una macchina a stati finiti. Ad ogni transizione da uno stato all'altro corrispondondono due *callback* (una che segnala l'uscita da uno stato e una che segnala l'entrata in un altro stato). Ad ogni *callback* corrisponde:

- un evento Flex che può essere intercettato attraverso un *listener* nel Bubble Module.
- una funzione Java del Bubble Controller.

Sia il Bubble Module che il Bubble Controller sono figli di altre classi che implementano già le funzioni e i *listener* d'uso comune. Affinché la parte Flex e la parte Java possano comunicare, è importante che l'inizio e la fine di ogni stato venga notificata al sistema attraverso opportune funzioni in Actionscript che ogni BubbleModule ha (ad esempio `notifyLevelStart`, `notifyLevelEnd` ecc.).

Per un maggior dettaglio sugli stati di una bubble si veda la sezione 6.3.3. Entrare più nel dettaglio sull'aspetto di implementazione esula comunque dagli scopi di questa tesi e verrà trattato nel manuale per gli sviluppatori Bubbles al momento in fase di stesura.

Una volta terminata la stesura del codice, è possibile creare il client *standalone* e il file *zip* da mettere online all'interno dell'applicazione Ri-Di semplicemente utilizzando due opportuni *task ANT* contenuti all'interno del file `build.xml` del proprio progetto Bubbles.

Quanto detto dovrebbe bastare per dare la possibilità di creare nuove bubble. Sicuramente però avere chiaro come la singola bubble si inserisce all'interno dell'intera infrastruttura Bubbles aiuta a sviluppare il codice più velocemente e con maggiore coscienza di ciò che si sta facendo. Nella prossima

sezione verrà dunque mostrata l'intera struttura di Bubbles e come le varie parti interagiscono tra loro.

6.3.3 Struttura del Framework Bubbles

Si è visto nella sezione precedente come sviluppare un nuovo software riabilitativo con il Framework Bubbles. Verrà ora mostrata la struttura generale di tutta l'infrastruttura Bubbles dove tale software riabilitativo si colloca.

A gestire tutte le bubble è un'applicazione creata ad hoc dalla Cooperativa Anastasis e chiamata *Fresh*. Nella versione online del sistema, Fresh è un oggetto SWF avviato da Ri-Di.

La versione *stand-alone* è di fatto una versione ridotta di Ri-Di, dove l'utente ha solo la possibilità di gestire una piccola anagrafica e, per ogni utente inserito, avviare Fresh. Il tutto appare all'utente come una normale applicazione del proprio sistema operativo:

- Al posto del browser con cui si accede alla versione online viene usato un mini browser utilizzando le librerie SWT e XULRunner (l'utente utilizzatore non ha neanche la percezione di star usando un browser).
- Come *servlet container* viene utilizzata una versione *embedded* di Jetty.
- Come *database server* viene utilizzato H2, un database SQL *lite embedded*.

Andando più nello specifico nella struttura, Fresh è composto da:

- `Fresh client`. È la parte in Flex, che mostra le bubble attive per l'utente e che, su richiesta, le carica e le esegue.
- `Bubbles Service`. È il webservice in Java che fa da riferimento verso l'esterno tra la parte logica e la parte presentazionale.
- `Orchestrator`. È la parte Java che si occupa di caricare la parte logica di ogni bubble, di costruire i percorsi riabilitativi da eseguire e di modi-

ficarli durante l'esecuzione dell'applicazione in base ai risultati ottenuti dall'utente e dalle regole del Sistema Esperto.

- `Mediator`. È la parte in Actionscript che si occupa di raccogliere i messaggi del Fresh client e delle bubble, di inviarle a Bubbles Service, di ricevere le risposte e di notificare i cambi di stato a tutti i componenti in gioco.

Le interazioni tra i vari componenti verranno descritte qui di seguito e mostrate (in un caso esemplificativo) nella figura 6.4.

Tutte le comunicazioni tra la parte Flash e la parte Java avvengono tramite richieste effettuate dal Mediator a Bubbles Service. Mediator concretamente effettua chiamate remote ai servizi esposti del webservice Bubbles Service tramite un *RemoteObject*. La comunicazione tra i due avviene tramite protocollo SOAP ed è resa trasparente agli sviluppatori tramite l'uso di Granite DS.

Il salvataggio dei dati, ad opera dell'Orchestrator e dei singoli Bubble Controller, viene effettuata utilizzando Serena Persistence e Serena Auth del sistema Serena.

Fresh è di fatto un motore a stati finiti: come per le singole bubble, così anche per Fresh ogni volta che il sistema transisce da uno stato all'altro, vengono chiamate due funzioni di *callback* del Mediator (una che segnala l'uscita da uno stato e una che segnala l'entrata in un altro stato). Il Mediator manda queste notifiche a Bubbles Service e così facendo avvia le operazioni specifiche per quella determinata situazione (ad esempio salvare i risultati parziali) e/o riceve i contenuti che verranno mostrati allo stato successivo.

Il diagramma degli stati di Bubbles è mostrato nella figura 6.3

Gli stati mostrati nella figura 6.3 sono i seguenti:

- *Loading*. Il sistema carica tutti i dati che gli servono per partire (come ad esempio quelli relativi all'utente che ha effettuato l'accesso).

- *Introduction*. Il *trainer* dà il benvenuto all'utente e gli mostra la sua situazione corrente.

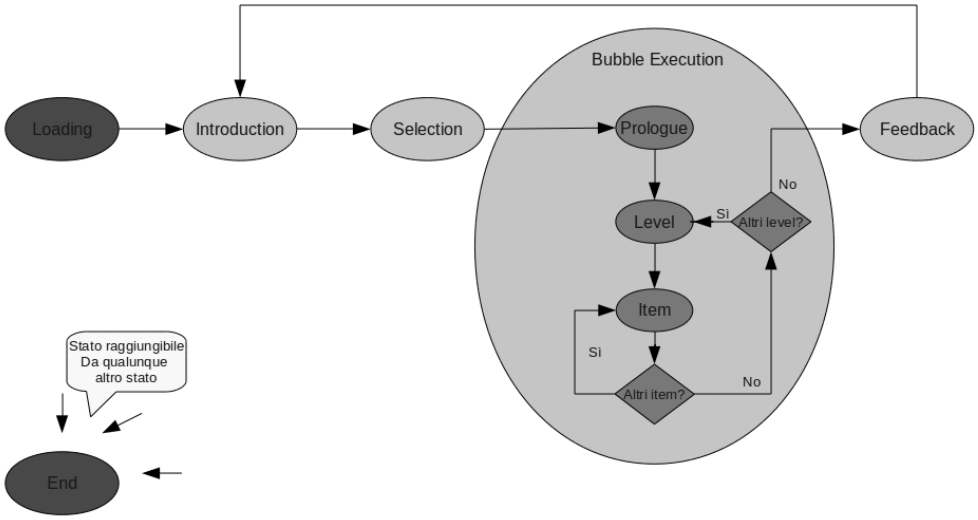

Figura 6.3: Diagramma degli stati di Bubbles

- *Selection.* L'utente o il sistema (in base alla configurazione) sceglie quale bubble eseguire

- *Execution.* Viene eseguito l'esercizio. In base al comportamento della singola bubble, tale stato può a sua volta al suo interno avere un suo ciclo di vita, con i seguenti stati (tutti opzionali):

  - *Prologue.* Fase preliminare dove il *trainer* spiega la consegna all'utente e, in alcuni casi, gli chiede alcune informazioni preliminari necessarie per l'esercizio.

  - *Level.* Viene eseguito un livello, inteso nel senso classico di livello di gioco (ad esempio viene mostrato un brano da leggere).

  - *Item.* Viene mostrato un oggetto atomico del livello (ad esempio, del brano da leggere, viene mostrata una parola).

  - *Partial feedback.* Il *trainer* mostra all'utente un *feedback* relativo al livello appena svolto (comunicando se è andato bene o meno).

- *Feedback.* Il *trainer* mostra all'utente un *feedback* relativo a tutto l'esercizio eseguito.

- *Exit.* In qualsiasi stato, l'utente può decidere di uscire dal sistema. In tal caso il sistema mostra un saluto all'utente e poi chiude l'applicazione.

A titolo di esempio viene qui mostrata la sequenza di azioni in seguito alla fine di una bubble:

1. Bubble Module notifica al Mediator la fine della bubble, passandogli i risultati ottenuti dall'utente.

2. Il Mediator gira la notifica e i risultati a Bubbles Service.

3. Bubbles Service gira la notifica e i risultati al corrispondente Bubble Controller.

4. Bubble Controller valuta i risultati ricevuti ed eventualmente aggiunge ulteriori dati ricavabili (ad esempio la velocità di esecuzione della bubble, analizzando il tempo di inizio e il tempo di fine già presenti all'interno dei risultati).

5. Bubble Controller crea la richiesta di salvataggio dei risultati e la invia a Serena Persistence.

6. Serena Persistence salva i risultati nel database

7. Bubble Controller analizza i risultati e crea conseguentemente il feedback da mostrare all'utente

8. Bubble Controller risponde a Bubbles Service comunicandogli l'avvenuto salvataggio e passandogli il feedback creato

9. Bubble Controller chiede all'Orchestrator quale sarà la prossima bubble da eseguire.

10. L'Orchestrator chiede a Serena Persistence la storia clinica del paziente, i risultati più recenti e le scelte effettuate dal clinico.

11. Serena Persistence legge i dati da database e li ritorna all'Orchestrator.

12. L'Orchestrator gira i dati al Sistema Esperto.

13. Il Sistema Esperto, in base alle proprie regole e ai dati ricevuti, decide quale dovrà essere la prossima bubble da eseguire e la comunica all'Orchestrator.

14. L'Orchestrator risponde a Bubbles Service comunicandogli quale dovrà essere la prossima bubble.

15. Bubbles Service risponde al Mediator, comunicandogli il feedback da mostrare e quale dovrà essere la prossima bubble.

16. Il Mediator notifica a Fresh Client e alla bubble che il server ha risposto correttamente e che tutte le azioni da effettuare in seguito alla fine della bubble sono state eseguite.

17. Bubble Module si chiude.

18. L'intero sistema passa allo stato di Feedback.

Tutto il processo è riassunto nel *sequence diagram* della figura 6.4.

Si conclude così la descrizione (necessariamente parziale) degli sviluppi futuri principali del sistema Serena, orientati principalmente a un miglioramento dell'usabilità del *framework* per chi sviluppa applicazioni e allo sviluppo del Framework Bubbles, un nuovo framework, figlio di Serena e specializzato all'esecuzione di programmi riabilitativi online autoadattivi.

Nel prossimo capitolo verranno tratte le conclusioni finali sul sistema Serena.

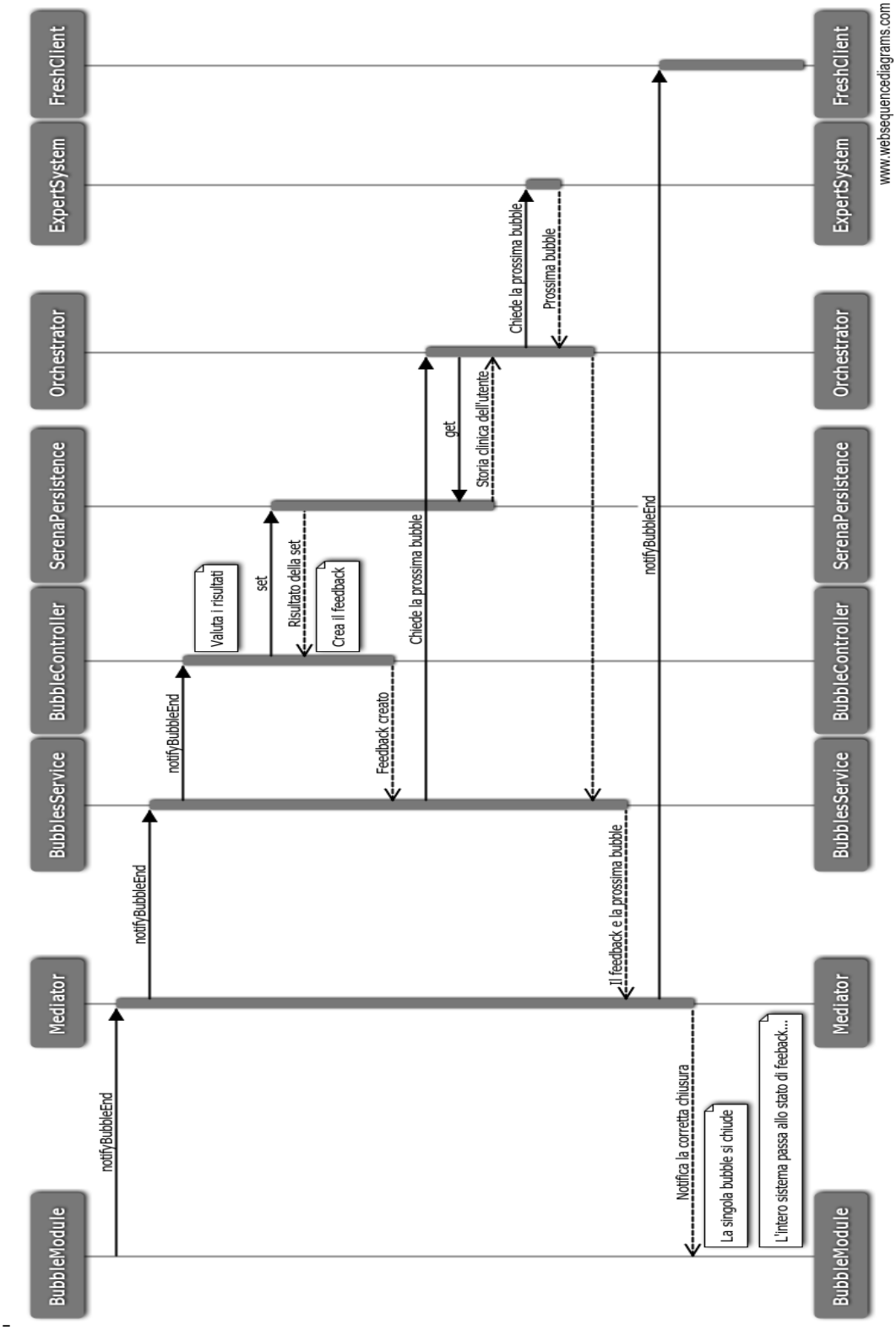

Figura 6.4: Sequence diagram per la fine di una bubble

Capitolo 7

CONCLUSIONI

In questa tesi è stata svolta la prima analisi completa di Serena.

Nel capitolo 1 è stato analizzato lo stato dell'arte in termini di framework di sviluppo, di applicazioni basate su ontologie, di sistemi *middleware* e di *application server* presenti sul mercato e in ambito di ricerca.

Nel capitolo 2 è stato mostrato il Serena Framework e il suo utilizzo da parte di uno sviluppatore di applicazioni Serena.

Nel capitolo 3 è stata analizzata nel dettaglio la struttura del Serena Application Server, all'interno del quale vengono eseguite le applicazioni Serena sviluppate con il Serena Framework.

Nel capitolo 4 è stata approfondita la robustezza di tutto il sistema e la sua conformità alle leggi vigenti.

Grazie a tale analisi, questa tesi ha potuto evidenziare luci e ombre di Serena, mostrate nel capitolo 5.

Si è quindi proposto nel capitolo 6 un insieme di sviluppi futuri per valorizzare i lati positivi di Serena e ridurre o annullare quelli negativi.

Nell'introduzione di questa tesi sono stati mostrati gli obiettivi che la Cooperativa Anastasis si prefiggeva quando è stato avviato il progetto Serena. Li riassumiamo qui, aggiungendo gli ulteriori obiettivi che si sono aggiunti in corso d'opera, quando Serena è diventato qualcosa di più ampio rispetto all'idea iniziale:

- Creare un framework di sviluppo di applicazioni web con un'insieme di automatismi che permettesse uno sviluppo veloce, soprattutto della prima versione prototipale di un'applicazione.

- Creare un sistema basato su ontologie, affinché la modellazione della base di dati fosse alla portata anche di chi non ha grande dimestichezza con i database, il risultato fosse facilmente mostrabile ad un cliente non informatico e i dati fossero utilizzabili da un sistema esperto, utilizzato soprattutto (ma non esclusivamente) per il supporto alla diagnosi (Serena nasce per essere utilizzato soprattutto in ambito sanitario).

- Creare un framework che separi lo sviluppo della parte grafica da quella logica, permettendo anche di sviluppare intere applicazioni senza scrivere una riga di codice.

- Creare un sistema che renda inter-operabili differenti strumenti informatici che insieme gestiscano l'intera "presa in carico" di un paziente (dallo screening, alla diagnosi alla riabilitazione).

Con la versione 1.5 di Serena, rilasciata in coincidenza con la stesura di questa tesi, si può dire che gli obiettivi prefissati sono stati raggiunti: Serena è utilizzato dalla Cooperativa Anastasis per lo sviluppo di tutte le sue applicazioni ed è stato oggetto di alcuni corsi di formazione affinché venga utilizzato anche da altre aziende.

Questa tesi ha comunque evidenziato alcuni margini di miglioramento del sistema e grazie a queste valutazioni il team di sviluppo potrà focalizzare meglio i suoi interventi futuri, affinché la prossima versione di Serena sia ancora più soddisfacente.

Un ulteriore effetto concreto di questa tesi è riscontrabile nel Framework Bubbles, descritto nella sezione 6.3.

Bubbles è un nuovo framework sviluppato dalla Cooperativa Anastasis e orientato allo sviluppo di software riabilitativi per persone con Disturbi Specifici dell'Apprendimento. È stato sviluppato tenendo in considerazione

i lati negativi e positivi di Serena emersi durante la stesura di questa tesi e attualmente si trova in fase di *beta testing*.

RICONOSCIMENTI

Come già scritto nell'introduzione, il sistema Serena è stato sviluppato dal team di sviluppo della Cooperativa Anastasis. Per la precisione, qui di seguito viene dato riconoscimento ai principali sviluppatori di ogni singolo componente del sistema:

- Serena Application è stato progettato e sviluppato da Andrea Frascari, Vincenzo Carnazzo, Andrea Pegoretti e Matteo Tassetti.
- Serena Auth è stato progettato e sviluppato da Andrea Frascari e Matteo Tassetti.
- Serena Persistence è stato progettato e sviluppato da Andrea Frascari.
- Serena Presentation è stato progettato e sviluppato da Vincenzo Carnazzo e Matteo Tassetti.
- Serena Node è stato progettato e sviluppato da Vincenzo Carnazzo.
- Serena Gateway è stato progettato e sviluppato da Vincenzo Carnazzo.
- Serena Axis è stato progettato e sviluppato da Andrea Pegoretti.
- Serena Module Object è stato progettato e sviluppato da Andrea Frascari, con la collaborazione di Vincenzo Carnazzo, Andrea Pegoretti e Matteo Tassetti.
- Serena Module Expert System è stato progettato e sviluppato da Nicola Azzini, con la collaborazione di Matteo Tassetti.

- Serena Module Login è stato progettato e sviluppato da Matteo Tassetti.
- Serena Module Menu è stato progettato e sviluppato da Matteo Tassetti.
- Serena Module Report è stato progettato e sviluppato da Vincenzo Carnazzo.
- Serena Module Stat è stato progettato e sviluppato da Matteo Tassetti.
- Serena Developer Tool è stato progettato e sviluppato da Matteo Tassetti.
- Template Generator è stato progettato e sviluppato da Vincenzo Carnazzo.
- PegoEditor è stato progettato e sviluppato da Andrea Pegoretti.
- La grafica dell'applicazione Serena predefinita è stata progettata e implementata da Andrea Bellocchio e Alice Pelliconi.

Serena fa inoltre uso delle seguenti librerie e delle seguenti applicazioni esterne:

- Apache Commons, della Apache Foundation, coperto da Apache License. È un gruppo di librerie con funzioni di base (per operazioni matematiche, gestione di stringhe, gestione di liste, accesso a database ecc.).
- Apache Lucense, della Apache Foundation, coperto da Apache License. È una libreria per creare motori di ricerca su testi.
- Apache Log4J, della Apache Foundation, coperto da Apache License. È una libreria per effettuare *logging* su file di testo.
- Apache Axis, della Apache Foundation, coperto da Apache License. È una piattaforma Java per creare e mettere online web service.

- Dom4J, di MetaStuff, coperta da BSD style License. È una libreria per leggere, navigare e manipolare documenti XML.
- Jaxen, di Bob McWhirter e James Strachan, coperto da Apache-like License. È un motore per XPath.
- Protégé Ontology Editor, dello Standford Center for Biomedical Informatics Research, è coperto da Mozilla Public License. È un editor di ontologie.
- Eclipse, della Eclipse Foundation, coperto dalla Eclipse Public License. È un IDE per lo sviluppo di applicazioni Java.
- Eclipse BIRT, della Eclipse Foundation, coperto dalla Eclipse Public LIcense. È un generatore di report.

Infine:

- Il Serena Eclipse plugin è stato progettato da Vincenzo Carnazzo, che attualmente lo sta sviluppando.
- Il Framework Bubbles è stato progettato e sviluppato da Vincenzo Carnazzo e Matteo Tassetti, in collaborazione con Enzo Ferrari, Fabrizio Piazza e Andrea Pegoretti.

Bibliografia

[Ana10] Cooperativa Anastasis. Pegoeditor. Website, 8 Novembre 2010. http://pegoeditor.anastasis.it.

[Azz08] Nicola Azzini. Integrazione di un motore inferenziale in un'architettura distribuita. Master's thesis, Università di Bologna, Facoltà di Ingegneria, 2007-2008.

[Bal09] Michael Ball. *Drools JBoss Rules 5.0 Developer's Guide*. Packt Pub Ltd, 2009.

[Bar03] Douglas K. Barry. *Web Services and Service-Oriented Architectures*. Morgan Kaufmann, 2003.

[Bec04] Kent Beck. *JUnit Pocket Guide*. O'Reilly Media, 2004.

[BK06] Christian Bauer and Gavin King. *Java Persistence with Hibernate*. Manning Publications, 2006.

[CLG07] Vivek Chopra, Sing Li, and Jeff Genender. *Professional Apache Tomcat 6*. Wrox Press, 2007.

[Coo10] Cooperativa Anastasis. *Serena 1.5, manuale per lo sviluppatore*, 2010.

[DR07] Snow Dowd and Robert Reinhardt. *Adobe Flash CS3 Professional Bible*. Wiley, 2007.

[GHRJ94] Erich Gamma, Richard Helm, RalphJohnson, and JohnVlissides. *Design Patterns. Elements of Reusable Object-Oriented Software.* AddisonWesley Professional, 1994.

[Gia97] Enrico Gianfelici. *La legge sulla privacy.* Fag, 1997.

[Goo00] James Goodwill. *Java Server Pages.* Apogeo, 2000.

[Guc10] Ceki Gucku. *The Complete Log4j Manual.* QOS.ch, 2010.

[Hen10] Greg Hendricks. Software testing with testopia. Website, 8 Novembre 2010. http://www.mozilla.org/projects/testopia.

[Hol04] Steve Holzner. *Eclipse.* O'Reilly Media, 2004.

[Hol05] Steve Holzner. *ANT, The Definitive Guide.* O'Reilly Media, second edition, 2005.

[HPT08] Nola Hague, Diana Peh, and Jane Tatchell. *BIRT. A Field Guide To Reporting.* Addison-Wesley Professional, second edition, 2008.

[Hun01] Jason Hunter. *Java Servlet Programming.* O'Really Media, second edition, 2001.

[IB02] Romin Irani and S. Jeelani Basha. *AXIS. Next Generation Java SOAP.* Peer Information, 2002.

[Kay00] Michael Kay. *XSLT Programmer's Reference.* Wrox Press, 2000.

[KRH+06] Andrew Kirkpatrick, Richard Rutter, Christian Heilmann, Jim Thatcher, and Cynthia Waddell. *Web Accessibility. Web Standards and Regulatory Compliance.* Friends of ED, 2006.

[LK08] Joey Lott and Chafic Kazoun. *Programming Flex 3.* Adobe Dev Library, 2008.

[Moo01] Colin Moock. *Actionscript. The Definitive Guide.* O'Reilly Media, 2001.

[oS10] University of Stanford. Protègè. Website, 8 Novembre 2010. http://protege.stanford.edu.

[Paw10] Renè Pawlitzek. Hamlets. Website, 8 Novembre 2010. http://hamlets.sourceforge.net.

[RB09] Graeme Rocher and Jeff Brown. *The Definitive Guide to Grails*. Apress, second edition, 2009.

[Sab95] Giovanni Sabbadini. *Manuale di neuropsicologia dell'età evolutiva*. Zanichelli, Bologna, 1995.

[Sca05] Roberto Scano. *Legge 04/2004: dalla teoria alla realtá*. IWA Italy, 2005.

[Sim02] John E. Simpson. *XPath and XPointer*. O'Really Media, 2002.

[Tad09] Lino Tadros. *TestComplete 7 Made Easy*. Falafel Software Inc., second edition, 2009.

[TVM+03] Sameer Tyagi, Michael Vorburger, Keiron McCammon, Heiko Bobzin, and Keiron McCannon. *Core Java Data Objects*. Prentice Hall, 2003.

[W3C01] W3C. Web content accessibility guidelines (WCAG) 2.0. Website, 8 Novembre 2001. http://www.w3.org/TR/WCAG20/.

[WD10] Franck Wolff and William DraÃ¯. Granite data services documentation. Website, 24 novembre 2010. http://www.graniteds.org/confluence/display/DOC.

RINGRAZIAMENTI

In questa tesi sono stati sviscerati tutti gli aspetti di Serena. Tutti eccetto uno.

Serena è soprattutto il frutto del lavoro del miglior team di sviluppo che potessi incontrare lungo la mia strada lavorativa, che mi ha fatto sentire a casa fin dal primo giorno di lavoro e che mi dà ogni giorno un esempio concreto di come dovrebbe essere ogni ambiente di lavoro ("Non privi di difetti ma ci compensiamo"). Un sincero grazie quindi (in rigido ordine alfabetico) ad Andrea Frascari, Andrea Pegoretti, Alice Pelliconi e Matteo Tassetti (i "serenizzatori"), a tutte le persone che compongono quella splendida azienda che è la Cooperativa Anastasis, alle nuove leve e a tutte le persone che, direttamente o indirettamente, hanno contribuito a Serena (la Apache Foundation *in primis*: non so come avremmo fatto senza di voi).

Inoltre un sentito grazie al mio relatore Claudio Sacerdoti Coen, per aver sviscerato Serena con competenza e puntigliosità, pur rispettando la mia particolare situazione di studente-lavoratore-padre.

Questa tesi deve comunque molto anche ad altre persone non coinvolte direttamente in Serena.

A Maura, sicuramente, per non essere mai stata gelosa di Serena e per essermi stata accanto in tutti i momenti in cui ho pensato che non sarei mai arrivato a scrivere la parola *fine* sulla mia carriera universitaria. E per esserci. Sempre.

A Sofia, per non aver distrutto questa tesi e non averne colarato i fogli (almeno spero). Per aver rallentato drasticamente la mia carriera universita-

ria ma anche per averle dato, insieme a Maura, un senso (come al resto della mia vita).

Ai miei genitori, perché se sono come sono è anche merito (e colpa) vostra.

Alla mia famiglia allargata (anche a Pamma!) per avermi fatto sempre sentire a casa.

Ad Emergency, alle Mondine di Novi, all'ARCI, all'ANPI e a tutti quelli che rendono questo mondo migliore.

Ad Alan Bertossi per avermi insegnato la perseveranza (all'amm'...).

Ad amiche ed amici vecchi e nuovi: considerateci ringraziati tutti (questa tesi si è dilungata fin troppo! E poi rischierei di dimenticare qualcuno).

Un grazie particolare a Pollicino, alle Ferrovie Emilia Romagna e ai suoi treni, senza i quali non sarei mai riuscito a scrivere alcunché.