Università degli Studi di Padova

Facoltà di Ingegneria

CORSO DI LAUREA MAGISTRALE
IN
INGEGNERIA INFORMATICA

TESI DI LAUREA

Integrazione dei framework Spring e Flex tramite Spring BlazeDs Integration

RELATORE: CH.MO PROF. MICHELE MORO

TUTOR AZIENDALE: DOTT. STEFANO FARDIN

Ringraziamenti

Voglio ringraziare il professor Michele Moro per avermi indirizzato nella scelta del progetto e per il supporto che, in qualità di relatore, mi ha dato nel redigere questo lavoro.

Ringrazio il mio tutor aziendale Stefano Fardin che mi ha seguito nello svolgimento dello stage indirizzandomi nello studio delle tecnologie più promettenti e coinvolgendomi nelle scelte progettuali.

Ringrazio l'azienda S.I.Pe. S.r.l. per l'ospitalità e per l'ottimo rapporto instaurato con i dipendenti, in particolare con i "colleghi" del team OpenDedalo: Alessandro, Gil, Michele e Riccardo.

Ringrazio tutti coloro che mi sono stati vicini nei momenti più difficili della mia vita e dedico questo lavoro alla mia famiglia e ai miei amici.

Indice

1 Introduzione

1.1 L'azienda

S.I.Pe. Srl, Soluzioni Informatiche Personalizzate, è una società con sede a Scorzè che opera nel settore dell'Information Technology. L'area di competenza primaria è il settore logistico-amministrativo.

L'azienda nasce come società di consulenza, ma negli anni ha sviluppato dei prodotti propri, che possono essere adattati e personalizzati per rispondere alle esigenze dell'utente. Questi prodotti subiscono un continuo aggiornamento sia tecnologico che funzionale.

L'azienda, nata nel'1984, è Business Partner IBM ed usa come tecnologia di riferimento il linguaggio Ile RPG-400 su piattaforma IBM System-i. In questi ultimi anni per adeguarsi all'evoluzione del mercato si sta sta indirizzando verso i temi del networking aziendale, della comunicazione, della continuità di servizio e del business intelligence.

1.2 L'attività di stage

Uno dei prodotti più utilizzati e apprezzati di S.I.Pe. è Dedalo. Questo è un sistema informativo per la gestione del magazzino sviluppato in Ile RPG-400 su piattaforma IBM System-i.
Dedalo raccoglie l’insieme delle funzioni necessarie alla gestione informatica dei flussi fisici del magazzino, intervenendo dalla fase di ricevimento merce fino a quella del carico per la spedizione. Non è stato sviluppato solo come soluzione alle problematiche operative, ma soprattutto come uno strumento per ottimizzare la gestione della struttura logistica delle aziende. Dal punto di vista funzionale è stato studiato per essere aderente alle necessità delle aziende che operano in ambienti di distribuzione, di produzione e di servizi logistici per conto terzi.

Per adattarsi all'evoluzione delle tecnologie informatiche, nell'Ottobre 2010 è stato deciso di svilupparne una nuova versione open-source in linguaggio Java. Questa operazione tuttavia non è stata un semplice porting della versione precedente. Infatti sono state eseguite delle nuove analisi funzionali che hanno portato a nuovi sviluppi.

Il nuovo prodotto, chiamato OpenDedalo, è ormai giunto alla fase di test presso il cliente. Tecnicamente è strutturato come una web application e implementa il pattern Model View Controller, che è ormai uno standard per le applicazioni web. Per il presentation layer è stato utilizzato il framework della Adobe Flex, che è nativamente predisposto a produrre un thick client molto efficiente ed elegante rispetto ad altre tecnologie analoghe. Dato che Flex è una tecnologia relativamente recente - nasce nel Marzo 2004 - c'è stato, e continua tuttora ad esserci, un grande sviluppo per quanto riguarda le modalità di integrazione con altri framework.

Per essere più precisi Opendedalo è stato pensato per tre tipologie di utenti diversi e quindi con tre diverse modalità di acceso. Infatti può essere utilizzato collegandosi attraverso un browser con un normale pc, con un terminale mobile o attraverso l'uso di un dispositivo a radiofrequenza, sostanzialmente delle cuffie e un microfono. Flex è stato utilizzato per realizzare il front-end dato che è particolarmente adatto per applicazioni di tipo mobile, come viene illustrato nel seguito.

Lo scopo principale dello stage è stato ottimizzare l'integrazione tra il framework Spring e Flex. Infatti inizialmente Flex non era ben integrato all'interno del framework Spring e per questo motivo la sua configurazione era alquanto gravosa e poco intuitiva. In prima approssimazione possiamo dire che la comunicazione tra Flex e Spring viene gestita dalla libreria BlazeDS di Adobe. Il lavoro dello stage è stato principalmente studiare quali vantaggi offre la nuova libreria Spring BlazeDS Integration rispetto all'uso della sola libreria di Adobe e determinare quali rischi o regressioni possa introdurre, con particolare attenzione alle prestazioni e alla sicurezza.

La seconda parte dello stage è stata invece incentrata completamente sul front-end Flex. Infatti anche se Flex nasce per applicazioni di piccole dimensioni porta con se alcuni problemi relativi soprattutto all'utilizzo della memoria che devono essere gestiti in applicazioni di più ampio respiro. Per questo motivo è stata studiata una nuova architettura del front-end che permettesse un uso più efficiente della memoria e che al contempo permettesse di riutilizzare un larga parte il codice già sviluppato.

Quindi l'attività dello stage ha comportato tre fasi. Nella prima parte ho studiato i framework utilizzati in OpenDedalo, quindi principalmente Spring, Hibernate e Flex. In parallelo a questo ho studiato l'architettura del sistema, ovvero i pattern architetturali utilizzati

nell'applicazione. Dopo questa prima fase di studio mi sono concentrato sulle potenzialità offerte dalla libreria Spring BlazeDs rispetto alla libreria di Adobe e, per studiarne le caratteristiche, ho sviluppato un mini-progetto con la stessa struttura di OpenDedalo. Dopo aver eseguito alcune prove su questo progetto la stessa impostazione è stata riportata sul progetto reale.

Nell'ultima fase dello stage ho studiato alcuni pattern comportamentali utili a risolvere il problema delle prestazioni e con l'aiuto del tutor aziendale è stata proposta una modifica all'architettura del front-end che risolvesse il problema dell'occupazione eccessiva di memoria.

2 Le tecnologie utilizzate

Gli strumenti utilizzati durante lo stage sono stati molteplici spaziando da ambienti di sviluppo a strumenti per l'amministrazione del database, per gli schemi UML o per il versionamento. Tra i più importanti, che vengono approfonditi in seguito, ci sono stati l'Ide Eclipse, il relativo plugin Flex Builder e i framework Spring e Hibernate e ovviamente le librerie Adobe BlazeDS e Spring BlazeDS Integration.

2.1 Spring

Spring è un framework open source per lo sviluppo di applicazioni Java. Permette di costruire applicazioni utilizzando “plain old java object” (POJO) applicandovi i servizi enterprise in maniera non invasiva secondo il modello di programmazione di JAVA SE e EE[1].

Parlando in generale Spring si propone di demandare alcune responsabilità dello sviluppatore al framework sia offrendo dei package che permettono una integrazione semplice delle funzionalità più diffuse nel mondo enterprise (ad esempio i package DAO, ORM, WEB), sia promuovendo l'uso di Best Practice come la programmazione verso interfacce piuttosto che verso classi, mantendo bassa la dipendenza dell'applicazione dalle API del framework e spostando la configurazione dei vari componenti in file di contesto XML. Quest'ultima impostazione insieme all'uso dell'Inversion of Control permette di evitare la proliferazione di

1 Concettualmente J2SE è un sdk per lo sviluppo di applicazioni in linguaggio Java, mentre Java EE è un insieme di tecnologie orientate a semplificare la scrittura di applicazioni entrerprise tramite componenti standardizzati, modulari e riutilizzabili.

classi singleton, che complicano i test, mantenendo semplice la manutenzione della configurazione dei vari aspetti dell'applicazione.

Spring nasce nel 2002 come allegato al libro Expert One-on-One J2EE Design and Development di Rod Johnson è ad oggi un framework maturo e molto diffuso nella comunità Java. Il libro di Johnson promuove un modello di sviluppo semplice in antitesi a framework invasivi, come gli EJB, secondo il motto “J2EE should be easier to use”. A differenza di questi Spring si basa sull'utilizzo di POJO senza obbligare le classi a seguire dei contratti spesso inutilmente complessi.

Spring è basato su una architettura a strati ed offre una integrazione semplice con le tecnologie più diffuse. Questo permette al progettista di scegliere quali moduli utilizzare integrandoli con altri prodotti senza essere vincolato a utilizzare il framework nella sua interezza. Inoltre ogni modulo è stato disegnato tenendo a mente la scrittura di unit test. Per quanto ritenuti una best practice, spesso non vengono eseguiti, perché comportano di fatto una riscrittura del codice. Spring invece, grazie alla programmazione tramite interfacce e all'IoC (Inversion of Control) rende questa operazione semplice e più circoscritta, di fatto comportando solo la modifica di pochi file di configurazione e non del codice vero e proprio.

Nello schema seguente viene mostrata la struttura del framework.

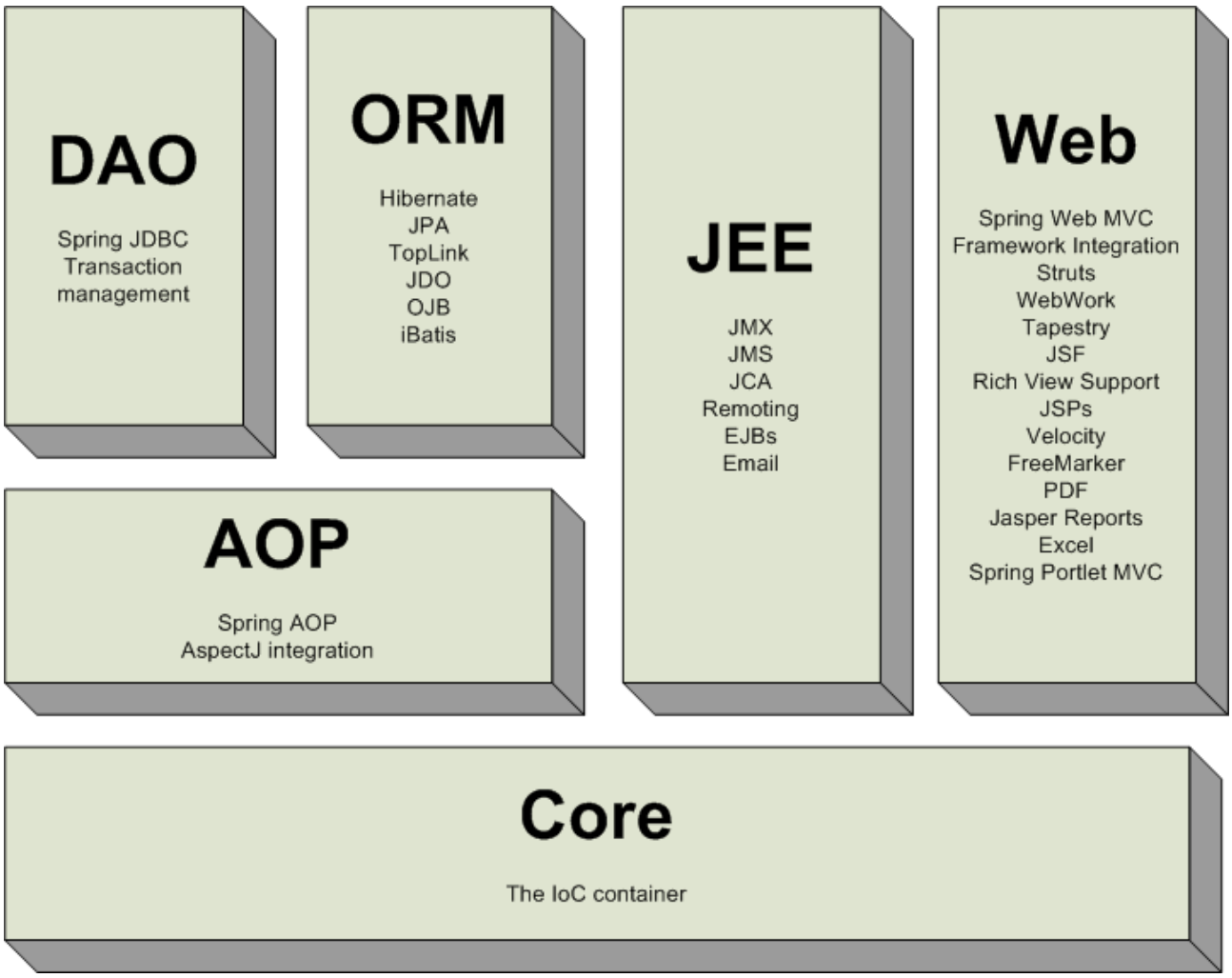

Figura 1: Overview del framework Spring

Tra i moduli che lo compongono possiamo trovare:

- Inversion of Control Container: un componente che gestisce la creazione e la risoluzione delle dipendenze di un Bean utilizzando il pattern IoC. Spring è un "lightweight container", perchè i Bean non devono aderire a nessun tipo di contratto[2] e possono essere rappresentati da una qualsiasi classe Java
- Programmazione Aspect-Oriented: un paradigma di programmazione basato sulla creazione di entità software che sovrintendono alle interazioni fra oggetti per realizzare funzionalità che coinvolgono più ambiti e che quindi provocherebbero una duplicazione del codice negli oggetti coinvolti.

2 Per essere più precisi un Bean deve esporre un costruttore senza argomenti, metodi get e set per ogni variabile accessibile all'esterno e implementare l'interfaccia Serializable.

- Data Access: tramite le API JDBC e strumenti di object-relational mapping è possibile utilizzare i più diffusi database relazionali e NoSQL
- Controllo delle transazioni: unifica diverse API per la gestione delle transazioni
- Model-View-Controller: implementazione del pattern MVC basato sul protocollo HTTP e sull'uso delle servlet.
- Convention-over-configuration: un paradigma di programmazione che prevede una configurazione esplicita da parte del programmatore solo per quegli aspetti che si differenziano dalle implementazioni standard.
- Batch Processing: un framework per l'esecuzione di processi batch di grosse dimensioni che comprende funzioni per il logging, tracing, gestione delle transazioni e il controllo della schedulazione.
- Sicurezza: un insieme di processi per il supporto all'autenticazione, autorizzazione e a vari protocolli e standard per la gestione della sicurezza.
- Remote Management: configurazione e controllo di oggetti Java locali e remoti tramite JMX.
- Messaging: una modalità di invio di messaggi tramite le API standard di JMS che permettte di registrare degli oggetti listener per un utilzzo trasparente del consumo dei messaggi tramite message queues.
- Testing: classi di supporto per la scrittura di unit e integration test.

Nel seguito vengono approfonditi i due aspetti del framework Spring che sono stati più rilevanti per il progetto svolto, ovvero l'inversione di controllo e il Model View Controller.

2.1.1 L'Inversion of Control o Dependency Injection

Il pattern di design Inversion of Control ha come fine produrre un codice fortemente disaccoppiato, leggero e adatto a essere sottoposto agli unit test. L'idea su cui si basa è che un oggetto invece di “procurarsi” da solo ciò di cui ha bisogno per funzionare si limiti a esporre le sue richieste tramite una qualche forma di contratto demandando così al container il

compito di fornirgli il necessario.

Per essere più precisi l'oggetto espone le proprie dipendenze, che comprendono sia le risorse del sistema sia altri oggetti. Questo porta alla creazione di un grafo di oggetti annidati in cui ciascun oggetto espone le proprie dipendenze all'oggetto chiamante. L'oggetto al livello più alto della gerarchia è quindi a conoscenza delle dipendenze di tutti gli oggetti utilizzati dall'applicazione. Questo oggetto top-level viene tipicamente utilizzato come entry point del sistema per assemblare tutti gli oggetti con le loro dipendenze prima di attivarli.

In un articolo ormai famoso del 2004 Martin Fowler metteva in evidenza come il termine Inversion of Control in realtà fosse troppo generico per denotare l'implementazione di questo pattern in Spring e proponeva invece di utilizzare il termine Dependency Injection (DI), che meglio ne esprime il funzionamento.

Per spiegare come funziona il pattern DI in Spring è utile seguire l'esempio classico di Fowler. Supponiamo di voler scrivere un componente che esegua una ricerca in un database e restituisca tutti i film girati da un certo regista. Questa funzionalità la implementiamo con un singolo metodo della classe MovieLister riportata in seguito.

```
class MovieLister {
    public Movie[ ] moviesDirectedBy( String myDir ){
    List allMovies = finder.findAll( );
    for ( Iterator it = allMovies.iterator( ); it.hasNext( ); ) {
        Movie movie = ( Movie ) it.next( );
        if ( ! movie . getDirector( ).equals( myDir ))
            it.remove( );
}
    return ( Movie[] ) allMovies.toArray ( new Movie [ allMovies.size( )]) ;
}
```

La classe è volutamente semplice, perchè l'oggetto di interesse è l'oggetto finder, anzi per

essere più precisi è il modo in cui la classe MovieLister viene collegata all'oggetto finder.

Quello che fa il metodo moviesDirectedBy non è altro che ottenere dall'oggetto finder tutti i film presenti sul database e eliminare quelli che non hanno il valore director uguale al parametro passato per la ricerca.

L'oggetto finder responsabile di accedere al database implementa l'interfaccia MovieFinder riportata in seguito.

```
public interface MovieFinder {
   List findAll( ) ;
}
```

Quello che succede senza usare il DI è che al momento della instanziazione della classe MovieLister le deve essere passato un riferimento a una classe che implementa questa interfaccia.

```
class MovieLister{
    …
   private MovieFinder finder;
   public MovieLister( ) {

 finder = new ColonDelimitedMovieFinder(‘movies1.txt‘);
}
```

L'ovvio problema di questa soluzione è che nel momento in cui si decide di usare una nuova implementazione dell'interfaccia MovieFinder si deve andare a modificare il codice. Già da questo esempio si vede come un approccio del genere comporta che scelte indipendenti dalle funzionalità del componente implichino delle modifiche al codice sorgente. Ovvero se, ad esempio, si decidesse di passare a una implementazione basata su un altro database si

dovrebbe modificare la classe MovieLister, senza che le sue funzionalità siano state intaccate da questa modifica.

Utilizzando il DI la classe potrebbe essere riscritta nel modo seguente:

```
class MovieLister{
   ...
   private MovieFinder finder
   public MovieLister(MovieFinder finder ) {
   this.finder = finder;
}


class ColonMovieFinder{
 ...
   private String fName;
   public ColonMovieFinder(String fName) {
   this.fName = fName;
}
```

La configurazione viene gestita in un file di contesto da Spring. In questo file troviamo indicata quale implementazione dell'interfaccia MovieLister deve essere utilizzata e anche quali parametri devono essere utilizzati nella creazione di questi oggetti.

```
<beans>
    <bean id="MovieLister "class="spring.MovieLister">
        <constructor-arg>
            <ref bean="MovieFinder" />
        </constructor-arg>
```

```
        </bean>
        <bean id="MovieFinder" class="spring.ColonMovieFinder">
            <constructor-arg>
                <value>movies1.txt</value>
            </constructor-arg>
        </bean>
    </beans>
```

In generale la dependency injection può essere realizzata in tre modi ovvero tramite costrutture, tramite metodi setter e tramite iniezione di metodi astratti. Quest'ultima modalità non è supportata da Spring.

Come si capisce del nome nella Constructor Injection il riferimento viene iniettato nel costruttore. Quindi nell'esempio del MovieFinder passerei l'implementazione del finder nel costruttore del lister ed il nome del file nel costruttore del finder:

```
class MovieLister{
   ...
private MovieFinder finder
   public MovieLister(MovieFinder finder ) {
      this.finder = finder;
}
class ColonMovieFinder{
   ...
   private  String fName ;
```

```
    public ColonMovieFinder(String fName ) {
        this.fName = fName ;
    }
```

Il file XML di configurazione conterrebbe quindi i tag seguenti.

```
<beans>
    <bean id="MovieLister" class="spring.MovieLister">
        <constructor-arg>
             <ref bean="MovieFinder"/>
        </constructor-arg>
    </bean>
    <bean id="MovieFinder" class="spring.ColonMovieFinder">
        <constructor-arg>
            <value>movies1.txt</value>
        </constructor-arg>
    </bean>
</beans>
```

In questo modo specifichiamo al container che stiamo utilizzando due bean MovieLister e MovieFinder che come costruttori hanno rispettivamente il finder e una Stringa che rappresenta il nome del file.

Nel Setter Injection la dipendenza viene passata come parametro di un metodo setter. È necessario utilizzare un costruttore vuoto e dei metodi setter per valorizzare le proprietà opportune. Quindi la class MovieLister dovrebbe essere modificata come nel codice seguente.

```
class MovieLister{
    ...
```

```
    private MovieFinder finder ;
    public void setFinder(MovieFinder finder){
        this.finder = finder ;
    }
}
class ColonMovieFinder {
    ...
    public void setFilename(String filename) {
        this.filename = filename;
    }
}
```

E l'XML di configurazione diventerebbe il seguente.

```
<beans>
    <bean id="MovieLister"class="spring.MovieLister ">
        <property name="finder">
            <ref local="MovieFinder"/>
        </property>
    </bean>
    <bean id="MovieFinder" class=" spring.ColonMovieFinder">
        <property name="filename">
        <value>movies1.txt</ value>
        </property>
    </bean>
</ beans>
```

Già da questi esempi molto schematici è possibile costatare l'utilità e la potenza del pattern Dependency Injection. Infatti le classi che andiamo a scrivere non implementano nessuna

interfaccia o classe astratta e non hanno nessuna dipendenza con l'Ioc Container.

Ogni modifica che andremo a fare, quindi, non impatterà in nessun modo sul codice già scritto, ma comporterà solo una modifica delle classi che implementano le interfacce utilizzate e dei file di configurazione. Da questo si capisce come sia possibile eseguire dei test nel modo più efficiente sostituiendo le classi “reali” con classi di test che implementano le stesse interfacce.

2.1.2 Il modulo Model View Controller

Lo scopo del pattern di design Model View Control (MVC) è di rendere il più possibile indipendenti e autonome tra loro le parti dell'applicazione adibite al controllo, all'accesso ai dati e alla presentazione secondo i principi del Low Coupling e dell'High Cohesion[3].

Questo permette di realizzare delle applicazioni in cui è possibile riutilizzare il model, ovvero lo strato che contiene la logica relativa ai business data, su viste diverse e in generale applicazioni nelle quali le modifiche restano isolate allo strato in cui vengono eseguite. Per esser più specifici il model contiene la rappresentazione dei dati dell'applicazione e le regole di business con cui tali dati vengono utilizzati e modificati.
La View è la vista del modello, ovvero l'interfaccia tra l'utente e l'applicazione.
Il controller interpreta le richieste della view traducendole in azioni che vanno ad interagire con il model aggiornando conseguentemente la view stessa.

Nell'ottica J2EE il pattern può essere implementato in questo modo:

- Model: java beans che incapsulano la logica applicativa e implementano l'accesso agli Enterprise Integration System (DBMS, Host, etc.);
- Controller: Servlet (o JSP) e classi dette RequestHandler per gestire le richieste dell'utente;

3 Con Low Coupling si intende sottolineare l'importanza del basso accoppiamento, e quindi delle scarse dipendenze, tra le classi in maniera tale che eventuali modifiche a una classe abbiano un impatto quanto più possibile limitato sulle classi dipendenti.
Con High Cohesion si sostiene invece un alto livello di isolamento, delle classi. Ovvero il fatto che una classe svolga il proprio compito quanto più possibile senza appoggiarsi ad altre classi. Quando una classe esegue troppi compiti diventa difficoltoso utilizzarla e manutenerla.

•View: pagine HTML (o JSP) che si occupano di gestire gli aspetti di rendering dei dati applicativi.

Il Controller è generalmente costituito da una Servlet che è una classe singleton che agisce da Façade effettuando sia le operazioni di controller che di dispatcher delle richieste proveniente dal client tier. Il suo compito è di creare l'opportuna classe RequestHandler a cui demandare la gestione della richiesta ed invocare l'opportuna vista. Deve quindi informare e aggiornare il Presentation Model, cioè l'insieme dei dati che vanno a comporre la vista finale per l'utente a fronte di modifiche che l'operazione di business, invocata dall'utente ed evasa dal RequestHandler, ha apportato ai dati dell'applicazione.

L'implementazione del pattern in Spring è abbastanza articolata e molto flessibile. Per capirne la struttura senza entrare nei dettagli dell'implementazione è utile seguire il seguente diagramma che mostra in che modo viene servita una richiesta generata dall'utente.

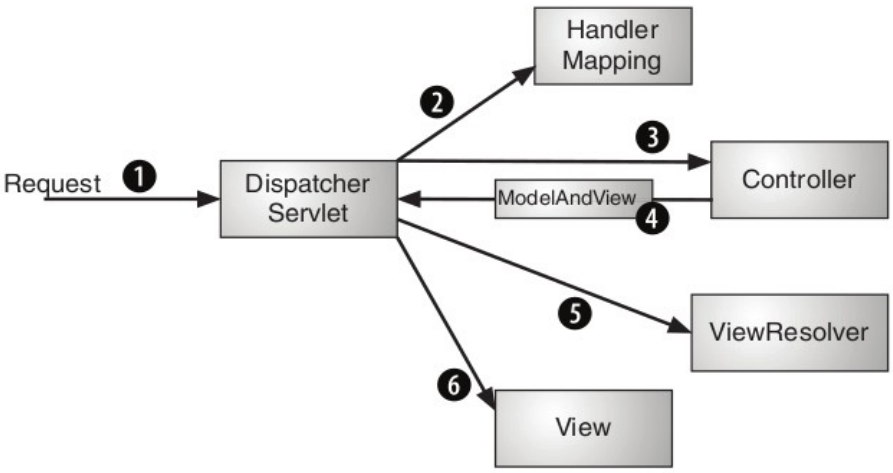

Figura 2: Il pattern MVC in Spring

Una request viene generata dal browser e porta con se varie informazioni necessarie all'esecuzione, tra queste informazioni c'è sempre almeno l'URL richiesto.

Inizialmente la richiesta viene passata al DispatcherServlet che è il FrontController dell'applicazione ed è quindi una classe singleton che non fa altro che smistare la chiamata ai componenti che eseguiranno l'elaborazione vera e propria. Quello che fa più in dettaglio è di mandare la request al controller. Tuttavia, dato che un'applicazione può avere più di un controller, il DispatcherServlet per fare questo interroga uno o più Handle Mapping, per

determinare quale deve gestire la richiesta.

Una volta inviata all'opportuno Controller la request gli fornisce tutte le informazioni necessarie porta e rimane in attesa che vengano elaborate. In un'applicazione reale il controller demanda l'elaborazione a delle classi specifiche per mantenere disaccoppiata la gestione del flusso di informazioni dalla logica di business dell'applicazione. Tipicamente l'esito dell'eleborazione del Controller è qualcosa che deve essere mostrato all'utente nel browser.

Quindi per presentare in modo comprensibile all'utente il risultato dell'eleborazione il Controller impacchetta i dati del model insieme al nome della view che deve essere utilizzate in un oggetto ModelAndView che viene rimandato al DispatcherServlet insieme all request processata.

Il nome della view contenuto nell'oggetto non è un riferimento all'implementazione della vista che deve essere utilizzata, ma un nome logico che viene risolto dal View Resolver nella pagina specifica.

Spring offre varie scelte per l'implementazione di ciascuno dei componenti sopra elencati con delle classi basi che possono essere estese e modificate a seconoda dai bisogni specifici dell'applicazione.

Tuttavia nella maggior parte delle applivazioni enterprise sono sufficienti le classi fornite dal framework in quanto è possibile configurarle in modo semplice tramite file di contesto XML.

2.2 Il Framework Flex

Flex è un framework open source prodotto dalla Adobe per la creazione di Rich Internet Application (RIA) per dispositivi mobili, web e desktop.

Fedele al motto “One basecode, multiple devices” consente di creare applicazioni per dispositivi che condividono una base di codice comune (oltre ai browser più diffusi supporta dispositivi con sistema operativo Android, BlackBerry Tablet OS, and iOS), riducendo i tempi e i costi di creazione delle applicazioni e di manutenzione. Adobe mette a disposizione gratuitamente un compilatore da riga di comando e a pagamento un ide (Adobe Flash Builder) che offre funzionalità avanzate quali la codifica intelligente, il debug passo-passo, il profiler

di memoria e prestazioni e la progettazione visiva.

Il modello di programmazione di Flex include sia i costrutti più diffusi nei linguaggi di programmazione attuali (tipizzazione forte, ereditarietà, interfacce, etc.) sia i componenti utilizzati nello sviluppo di applicazioni HTML (tabelle, layout, form, etc.), per questo esibisce una curva di apprendimento molto schiacciata soprattutto per gli sviluppatori cha hanno un background di sviluppo di applicazioni web.

Flex si integra facilmente con applicazioni lato server scritte in Java, PHP, Ruby, .NET, ColdFusion, and SAP tramite standad come REST, SOAP, JSON, JMS, and AMF. Fornisce, inoltre, un modello di messaggistica publish/subscribe e un servizio (Collaboration Service) per il supporto alla videoconferenza, al VOIP e in genere alle applicazioni che richiedono una comunicazione in real-time.

2.2.1 Un breve cenno storico: da Flash a Flex

Inizialmente i contenuti in formato Flash venivano creati utilizzando il programma di authoring multimediale Flash, che offriva oltre che supporto alla grafica vettoriale e bitmap il linguaggio di scripting ActionScript e streaming bidirezionale per contenuti audio e video.

Tra i punti di forza della tecnologia c'era la disponibilità di varie versioni del Flash Player per una vasta gamma di dispositivi oltre a pc e telefoni celluri.

Per essere più precisi possiamo distinguere tra l'ambiente di sviluppo integrato (IDE) Adobe Flash Professional e il Flash Player che è una macchina virtuale sulla quale vengono eseguiti i file Flash.

Il principale limite di Flash nasce dal fatto che venne progettato inizialmente per i grafici e questo si vede nell'uso del concetto di timeline che è estraneo ai linguaggi di programmazione più diffusi.

Per superare questo limite venne sviluppato Flex che nasce appunto per offrire degli strumenti per creare dei contenuti Flash comprensibili agli sviluppatori. Per fare questo Flex usa i concetti più familiari ai programmatori di workflow e di modello di programmazione al posto di quello della timeline.

Inizialmente Flex venne rilasciato come una applivazione J2EE o come tag library JSP che compilava un estensione dell'XML chiamata MXML e un linguaggio orientato agli oggetti chiamato Actionscript in applicazioni Flash. Il formato dei binari compilati era SWF, che sta per Small Web Format o Shockwave Flash.

Il punto di forza di Flash, e quindi di Flex, è di creare applicazioni web con “live data” o, per essere più tecnici, di permettere la creazione di un thick client che utilizzi meno interazioni con il server rispetto a delle pagine simili sviluppate usando JSP, ASP, o PHP.

Questo è reso possibile grazie all'integrazione nativa con la libreria LiveCycle Data Services. Questa mette a disposizione una suite di strumenti per la gestione dei dati in una web application scritta in Java. Offre “out of the box” una infrastruttura completa per il messaging tra applicazioni Flex che comprende servizi RPC, il Data Management Service e il Message Service. In breve i servizi RPC sono normali chiamate a procedura remota utilizzate per accedere a dati esterni. Il DMS fornisce la sincronizzazione in tempo reale in caso di aggiornamento dei dati, paginazione dei dati on-demand, replicazione e integrazione di dati diversi attraverso adapter. Nella pratica permette di creare applicazioni che lavorano con dati distribuiti e li manipolano come fossero delle relazioni annidiate. Il Message Service fornisce servizi di messaging per applicazioni real-time.

Questa libreria è codice proprietario e viene fornita sotto licenza Adobe, una sua versione “depotenziata” viene offerta come codice open source con la licenza LGPL con il nome di BlazeDS. Questa versione della libreria offre una parte significativa delle funzionalità della LiveCycle, ma sostanzialmente non offre supporto per le applicazioni real time e permette di gestire solo un numero limitato di comunicazioni contemporanee, quindi è sufficientemente potente per applicazioni web che non debbano offrire servizi real time e che abbiano un numero di utenti contenuto.

2.2.2 Adobe BlazeDS e Spring BlazeDs Integration

La libreria BlazeDs sviluppata da Adobe ha lo scopo di gestire la comunicazione in una applicazione client-server. A questo scopo mette a disposizione diversi servizi che permettono a più client[4] Flex di comunicare in modo sincrono o asincrono con un server J2EE.

4 Per essere più precisi il lato cliente è una combinazione di HTML, Flex e Javascript. Nell'uso più tipico però HTML e Javascript vengono utilizzati solo marginalmente. È possibile utilizzare la libreria con un client

La figura seguente mostra l'architettura di base.

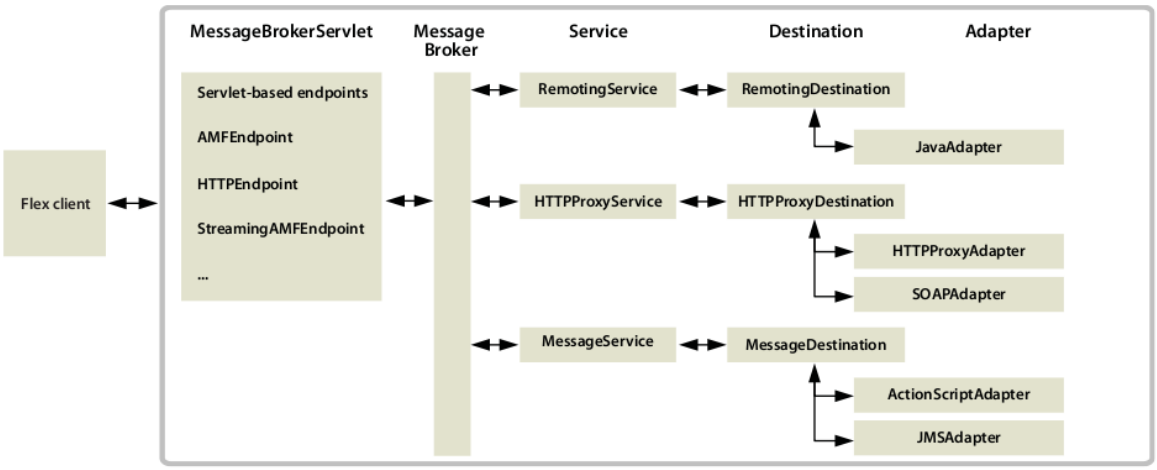

Figura 3: Struttura delle comunicazioni client-server con BlazeDs

Fondamentalmente BlazeDs permette due tipi di interazione: RPC (Remote Procedure Call) e Messaging Service. Le RPC permettono ai client di effettuare delle chiamate asincrone di servizi remoti che effettuano determinate operazioni sul server e restituiscono i risultati direttamente al client. I componenti RPC lato client, messi a disposizione nell'SDK, forniscono servizi HTTP (componente HTTPService), web service (componenti WebService) e remote object service (componenti RemoteObject). Senza l'ausilio di BlazeDS, usando solamente l'SDK di Flex, è possibile effettuare delle chiamate HTTP o di web service, ma non è possibile invocare direttamente metodi di oggetti Java. Una delle caratteristiche più interessanti è data proprio dal fatto che il componente RemoteObject (lato client) permette di accedere a oggetti Java senza la necessità di configurarli come web service. Il Messaging Service permette alle applicazioni client di comunicare fra di loro in maniera asincrona scambiando messaggi attraverso il server.

Uno dei principali vantaggi di BlazeDS è l'uso dell'AMF (Action Message Format) per la trasmissione dei dati tra Flex e Java. AMF è un protocollo binario supportato nativamente dal flash player e quindi il suo uso rende la trasmissione veloce ed efficiente. A rendere più agevole l'uso del protocollo è la definizione di livelli di astrazione superiori che permettono

sviluppato interamente in HTML e Javascript utilizzando AJAX per gestire la comunicazione al costo di una maggior complessità implementativa

di focalizzare l’attenzione sulla logica applicativa trascurando i dettagli della comunicazione.

La comunicazione attraverso la rete avviene utilizzando i Canali. Un canale incapsula i dettagli relativi al formato del messaggio e ai protocolli di rete. I due estremi del canale sono l'endpoint, ovvero il lato server e il destination, lato client. Quest'ultimo dal punto di vista pratico rappresenta l'identificatore dell'oggetto remoto che viene utilizzato da Flex.

Quindi, quando un oggetto remoto viene invocato è il canale ad essere responsabile di gestire il marshalling e unmarshalling dei dati. Dopo che è stato invocato il servizio opportuno, quindi è stato eseguito il codice Java, viene utilizzato un dispatcher, il messagebroker, che è responsabile di determinare a quale destination inviare la comunicazione.

Destinations, channels, endpoints sono appunto delle astrazioni che rappresentano concetti chiave del framework Flex. Le entità server side sono mappate su nomi logici e vengono invocate usando questi nomi. Quando un client interagisce con un servizio server-side tramite una destination, usa un canale di comunicazione specifico per le comunicazioni da e verso il server. Avremo ad esempio un AMF channel per gestire l’encoding e la comunicazione in AMF[5] e un HTTP channel per gestire l’encoding e la comunicazione in HTTP.

2.3 L'ide Eclipse

Eclipse è un ambiente di sviluppo integrato multi-linguaggio e multi-piattaforma. L'intero progetto è open-source e utilizzabile liberamente. Nasce da un consorzio di grandi società chiamato Eclipse Foundation che annovera tra i suoi componenti Ericsson, HP, IBM, Intel, MontaVista Software, QNX, SAP e Serena Software. Il progetto è sviluppato in Java ed è quindi disponibile per i più diffusi sistemi operativi. Oltre ad offrire un IDE completo per il linguaggio Java è possibile utilizzarlo per scrivere codice C++ e, grazie ad alcuni plugin, anche PHP, XML e Javascript.

La piattaforma di sviluppo è incentrata sull'uso di plugin, cioè di componenti software ideat per uno specifico scopo, per esempio la generazione di diagrammi UML, ed in effetti tutta la

5 Action Message Format è un formato binario realizzato dalla Adobe e introdotto a partire dal 2001 in Flash 6. E' un formato compatto per serializzre il grafo degli oggetti ActionScript senza duplicare oggetti identici o produrre riferimenti circolari.
Un oggetto di tipo AMF può poi essere usato per recuperare o far persistere lo stato di una applicazione tra due sessioni o per far comunicare in modo efficace due applicazioni. Stando alla documentazione ufficiale la comunicazione tramite AMF può essere fino a 10 volte più veloce rispetto al formato XML su SOAP.

piattaforma è un insieme di plug-in, versione base compresa. Nella versione base è possibile programmare in Java, usufruendo di comode funzioni di aiuto quali: completamento automatico ("Code completion"), suggerimento dei tipi di parametri dei metodi, possibilità di accesso diretto a CVS e riscrittura automatica del codice (detta di Refactoring) in caso di cambiamenti nelle classi.

Tra i plugin più importanti nello svluppo del progetto, oltre a quelli base per Java EE, ci sono stati il Flex Builder e il Flex Debugger. Anche se il Flex Builder non ha la stessa potenza dei plugin relativi a Java per quanto riguarda l'autocompletamento e i controlli sintattici durante la scrittura del codice si è rivelato molto utile per la possibilità di vedere in anteprima le pagine mxml durante la scrittura del codice. Oltre a questi sono stati fondamentali i plug-in per la gestione del versionamento insieme a Tortoise.

2.4 L'ORM tramite Hibernate

Hibernate è un noto strumento di ORM (ObjectRelational Mapping), che fornisce un framework Java per effettuare il mapping tra una modellazione ad oggetti ed un tradizionale database relazionale.

Il progetto Hibernate è opensource, nasce nel 2001 ad opera di Gavin King e anche se nel 2003 entra nell'offerta Jboss è ancora disponibile nella versione open. Lo scopo di un ORM è quello di gestire in modo automatico e trasparente la persistenza di oggetti Java, scrivendone le proprietà nelle apposite tabelle di un database relazionale. In particolare quello che Hibernate permette di fare è di usare delle classi Java che mappano le entità presenti nel database in modo trasparente. In alternativa infatti sarebbe necessario utilizzare oggetti generici e chiaramente ogni modifica al codice o alle classi comporterebbe grosse modifiche all'intero progetto. Tramite la mappatura tra classi e tabelle, che si esegue nel modo più semplice e standard utilizzando le annotation e il pattern del convention-over-configuratio, è possibile disaccoppiare le classi Java dalle tabelle che rappresentano rendendo il codice prodotto estremamente robusto.

Inoltre, il fatto che oggetti Java risiedono, o, per meglio dire, sono mappati in un database relazionale, rende possibile l'esecuzione di specifiche query sfruttando un particolare

linguaggio simile all'SQL, chiamato HQL (Hibernate Query Language), oppure tramite le cosiddette criteria queries, una variante “ad oggetti” dell'HQL.

Mappando una classe con Hibernate, si sta a tutti gli effetti delegando al framework la gestione delle sue proprietà e delle relazioni con le altre classi. Ad esempio, se una classe mantiene in un campo dati un riferimento ad un'altra, la relazione tra le due entità non avverrà tramite il campo riferimento (che rimarrà nullo), ma bensì sarà nel database stesso, sulle tabelle rappresentanti le due classi (con una chiave esterna, ad esempio, o con una join table,

dipendentemente dal tipo di relazione e dalle preferenze dell'utente). Qualora si acceda alla seconda entità tramite il riferimento contenuto nella prima, il sistema effettuerà automaticamente una query con una join tra le tabelle, per recuperare l'entità desiderata.

Quello che avviene in pratica è che attraverso dei file di configurazione xml si stabiliscono le corrispondenze tra gli oggetti usati all'interno dell'applicazione e le tabelle del db.

Hibenate si pone tra il modello ad oggetti della nostra applicazione e e il suo corrispondente esistente nel databse formato da tabelle, chiavi primarie e vincoli relazionali.

Hibernate rende persistenti praticamente ogni classe Java che lo richieda senza eccessivi vincoli. Ogni POJO (Plain Old Java Object) disegnato in fase di modellazione del dominio applicativo, per poter essere gestito da Hibernate deve semplicemente aderire alle fondamentali regole dei javabeans: cioè fornire metodi pubblici getter e setter per le proprietà da rendere persistenti. Una volta configurato correttametne il framework non è più necessario scrivere codice SQL per accedere al databse.

Il motore di Hibernate può occuparsi di tutto il lavoro necessario per estrarre, inserire e modificare i dati che costituiscono lo strato degli oggetti del nostro dominio applicativo generando autonomamente e in tutta trasparenza i comandi e le interrogazioni SQL necessari.

3 Il caso di studio

3.1 OpenDedalo

DEDALO è un sistema informativo per la gestione ed il controllo di tutte le attività di movimentazione di deposito. Raccoglie l'insieme delle funzioni necessarie alla gestione dei flussi fisici del magazzino intervenendo dalla fase di ricevimento merce fino a quella del carico per la spedizione.
Per le sue caratteristiche funzionali è aderente alle necessità di aziende appartenenti a vari rami merceologici, che operano in ambienti di distribuzione, produzione e di servizi logistici per conto terzi. Riscrivere il prodotto in Java ha comportato una nuova fase di analisi che ha permesso di aggiungere delle nuove funzionalità. I risultati della fase di analisi sono riassunti in seguito.

3.1.1 Obiettivi e Requisiti Architetturali

Le caratteristiche principali di OpenDedalo riprendono sia quelle della versione precedente, DEDALO, sia altre emerse da analisi funzionali.

Nel seguito le più importanti:

- indipendenza dalla piattaforma: il prodotto dovrà essere il più possibile libero da vincoli relativamente alla piattaforma di esecuzione, sia di tipo hardware, di sistema operativo, di database che di vendor
- accesso remoto: l'accesso ai servizi del prodotto deve poter avvenire da postazioni client remote rispetto al server centrale; su tali postazioni client deve inoltre essere ridotta il più possibile l'attività di installazione di software sia relativo al prodotto che di terze parti
- prestazioni: i tempi di risposta dell'interfaccia utente del nuovo prodotto devono essere il più possibile vicini a quelli della implementazione precedente in linguaggio Ile RPG-400 su piattaforma IBM System-i (tempi di risposta medi di qualche decimo di secondo motivati, oltre che dalla capacità elaborativa del server, anche da un limitato flusso di informazioni verso il client)

- estensibilità dell'interfaccia utente: la tecnologia client di pubblicazione dell'interfaccia utente deve consentire l'implementazione di logiche di interazione più o meno complesse (ad es. gestione di una griglia editabile)
- indipendenza dal database: il prodotto deve ridurre il più possibile la dipendenza dal vendor del database ed i più comuni motori relazionali devono essere supportati (ad es. Oracle, DB2, MySQL)
- sicurezza: l'accesso ai servizi applicativi è soggetto a politiche di autenticazione ed autorizzazione configurabili; il sistema deve prevedere inoltre la tracciabilità delle operazioni eseguite dagli utenti
- performance: il prodotto deve ammettere configurazioni in alta affidabilità, con clustering del server applicativo e load balancing delle richieste (con o senza affinità di sessione)
- configurabilità per utente: gli eventuali valori di default dei campi di ingresso delle mappe fanno parte del profilo utente collegato al sistema; si vuole poter attribuire queste proprietà per un gruppo di utenti semplificando la fase di configurazione (un nuovo utente associato ad un gruppo già censito eredita tutti i privilegi associati al gruppo); anche le colonne visibili di una data lista ed il loro ordine sono definite nel profilo utente
- accesso a documentazione in linea: l'utente può in ogni momento accedere a tre diversi tipi di documentazione in linea: quella relativa alla funzionalità scelta, quella della mappa correntemente visualizzata e quella specifica di un campo nella mappa; in tutti e tre i casi viene pubblicata una nuova mappa con la documentazione relativa
- servizio stampa: la generazione di uno spool può avvenire come esito di una iterazione diretta dell'utente (stampa interattiva) o come risultato di una elaborazione asincrona (stampa batch)
- lancio di servizi: l'utente ha la possibilità di far nascere nel sistema dei processi asincroni , che a loro volta potrebbero ulteriormente innescarne altri

3.1.2 Use-Case View

Gli attori del sistema derivati dai casi d'uso considerati sono:

- Amministratore, utente con privilegi di configurazione del sistema
- Utente Desktop, utente con privilegi d'uso del sistema a cui accede da una postazione client di tipo desktop
- Utente Radio, utente con privilegi d'uso del sistema a cui accede mediante un dispositivo a radio frequenza
- Utente Vocale, utente con privilegi d'uso del sistema a cui accede mediante un dispositivo a comando vocale
- Sistema Commerciale, sistema esterno con la responsabilità di mantenere i dati relativi alle anagrafiche principali (articoli e interlocutori)
- Clock di Sistema, dispositivo di temporizzazionedel sistema

Gli attori che vengono invece sollecitati dal sistema sono:

- Sistema Commerciale, sistema esterno con la responsabilità di mantenere i dati relativi alle anagrafiche principali (articoli e interlocutori)
- Stampante, dispositivo di stampa dei documenti prodotti dal sistema
- Directory Server, servizio di directory con accesso mediante LDAP per la gestione dell'anagrafica utenti

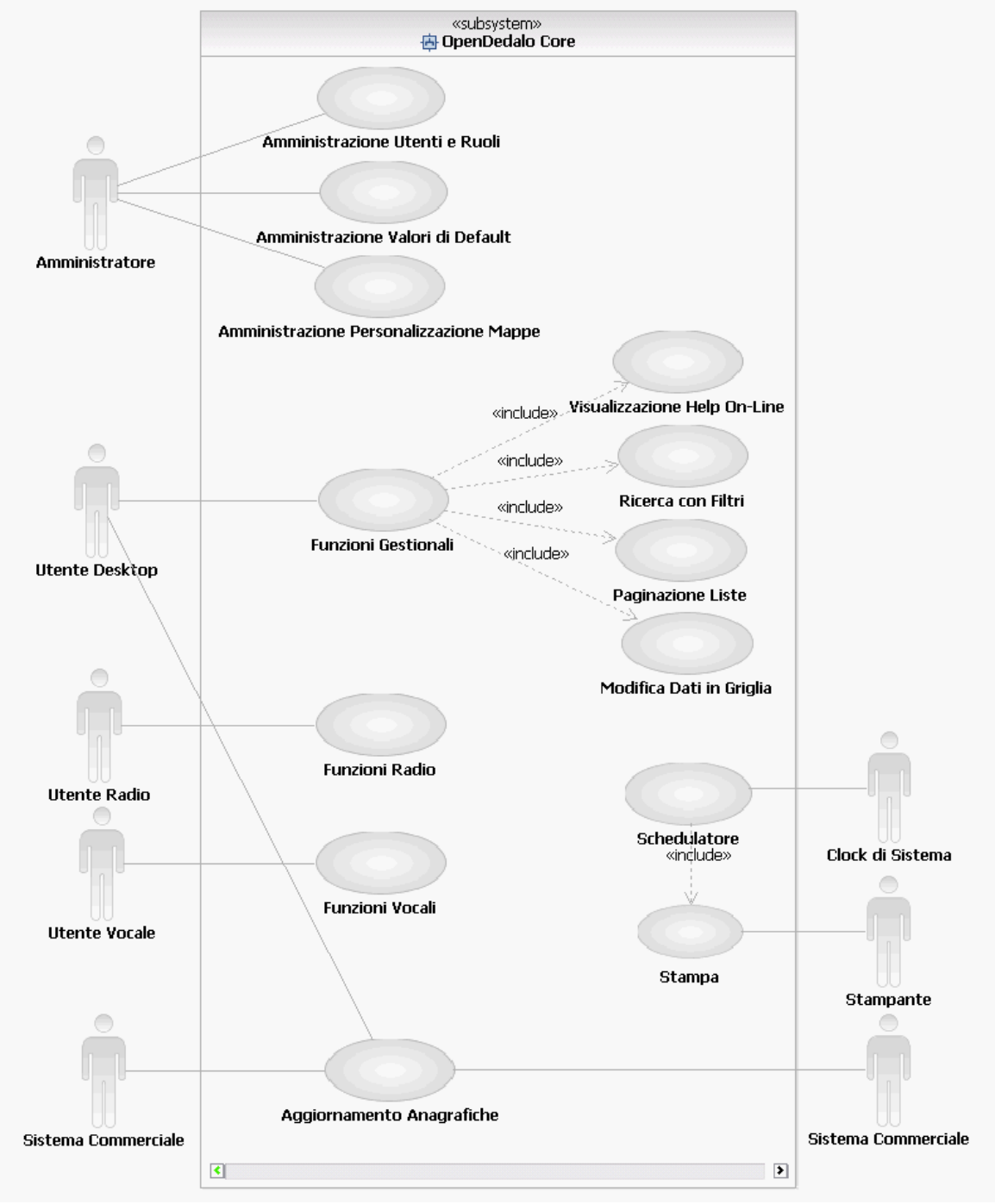

Figura 4: Struttura di OpenDedalo

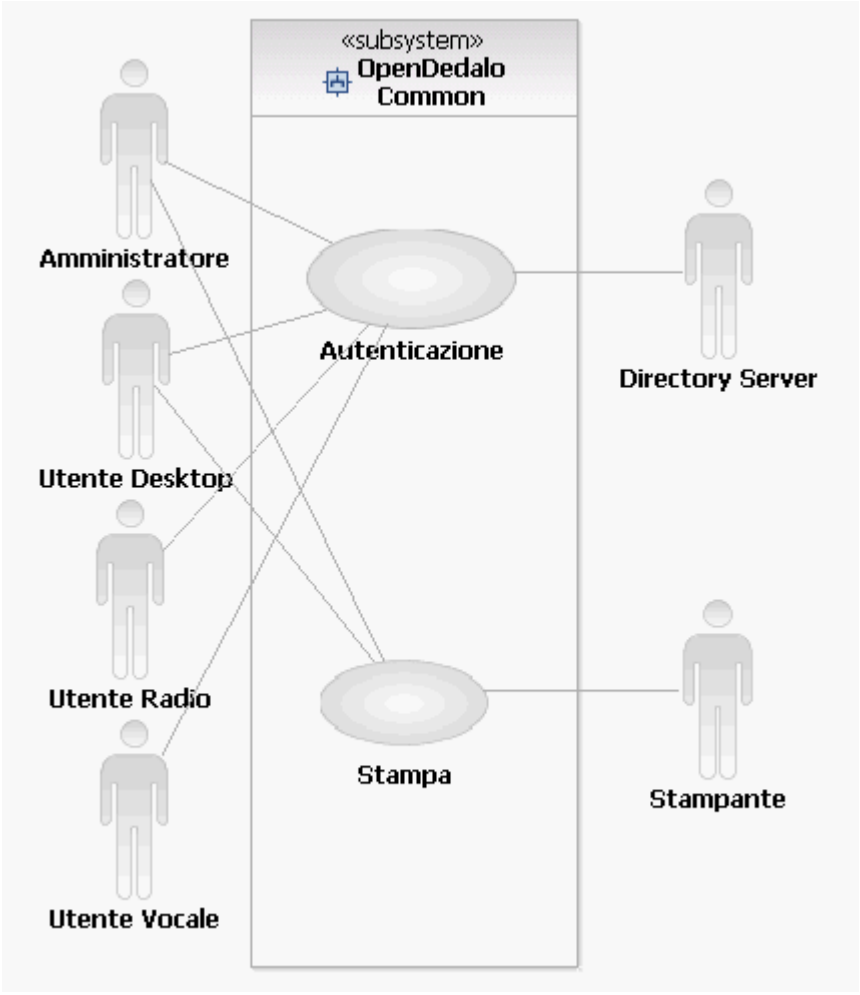

Figura 5: Casi d'uso di OpenDedalo

I seguenti casi d'uso sono considerati di impatto architetturale:

- Amministrazione Utenti e Ruoli, funzionalità di amministrazione per la definizione dei gruppi di utenti, dei ruoli associabili ai gruppi, e dei privilegi stabiliti per ogni ruolo nei confronti dei servizi del sistema
- Amministrazione Valori di Default, funzionalità di amministrazione per la definizione dei valori di default delle informazioni in input al sistema associati ad un gruppo di utenti
- Amministrazione Personalizzazione Mappe, funzionalità di amministrazione per la definizione delle informazioni che sono visibili, ed eventualmente modificabili, da un gruppo di utenti; anche le colonne di una data lista di informazioni che sono visibili o

meno per un gruppo di utenti sono configurabili

- Funzioni Gestionali, insieme delle funzionalità del prodotto rivolte agli utenti
- Visualizzazione Help On-Line, funzionalità di visualizzazione della documentazione in linea, a livello di funzione, pagina e campo
- Ricerca con Filtri, funzionalità di ricerca di un'informazione nel sistema specificando uno o più filtri
- Paginazione Liste, funzionalità di pubblicazione di una lista di informazioni mediante paginazione
- Modifica Dati in Griglia, funzionalità di editing di informazioni pubblicate su una griglia (ad es. una lista paginata) che permette la valorizzazione di più campi in input ad un servizio
- Funzioni Radio, funzionalità invocate dagli utenti che dispongono di un dispositivo a radio frequenza
- Funzioni Vocali, funzionalità invocate dagli utenti che dispongono di un dispositivo a comando vocale
- Schedulatore, funzionalità che prevede l'avvio automatico di attività del sistema secondo una schedulazione temporale configurabile (ad es. batch avviati periodicamente)
- Stampa, funzionalità di stampa di uno o più documenti prodotti dal sistema su una o più stampanti collegate
- Aggiornamento Anagrafiche, funzionalità di aggiornamento della eventuale copia locale delle anagrafiche gestite dal sistema commerciale (articoli e interlocutori ecc.)
- Autenticazione, funzionalità di verifica delle credenziali di un utente ed attribuzione di un profilo

Panoramica dell'architettura utilizzata

Dall'analisi dei requisiti funzionali e non del sistema, il modello che meglio risponde alle caratteristiche di distribuzione richieste è quello web, usando il browser come strumento di accesso ai servizi. I pattern architetturali per questo tipo di applicazioni sono:

- Thin Web Client, appropriato nel caso il nodo client abbia capacità computazionali minime o non si abbia il controllo sulla sua configurazione; in tal caso l'applicazione client è costituita da un browser per la pubblicazione di semplici pagine HTML (con eventualmente un limitato supporto a JavaScript) a basso contenuto interattivo
- Thick Web Client, appropriato nel caso il nodo client abbia buone capacità computazionali e si abbia un controllo minimo sulla sua configurazione; in tal caso l'applicazione client è costituita da un browser con un supporto completo per l'esecuzione di codice JavaScript (AJAX), VBScript, Java (Java plug-in, applet), ActiveX o Flash (Flash plug-in); l'interfaccia utente ha un alto contenuto interattivo

La tecnologia di riferimento scelta è Java Enterprise Edition (JEE) per l'implementazione, lato server, delle logiche applicative, ed è AJAX e Flash per l'implementazione, lato client, dell'interfaccia utente.

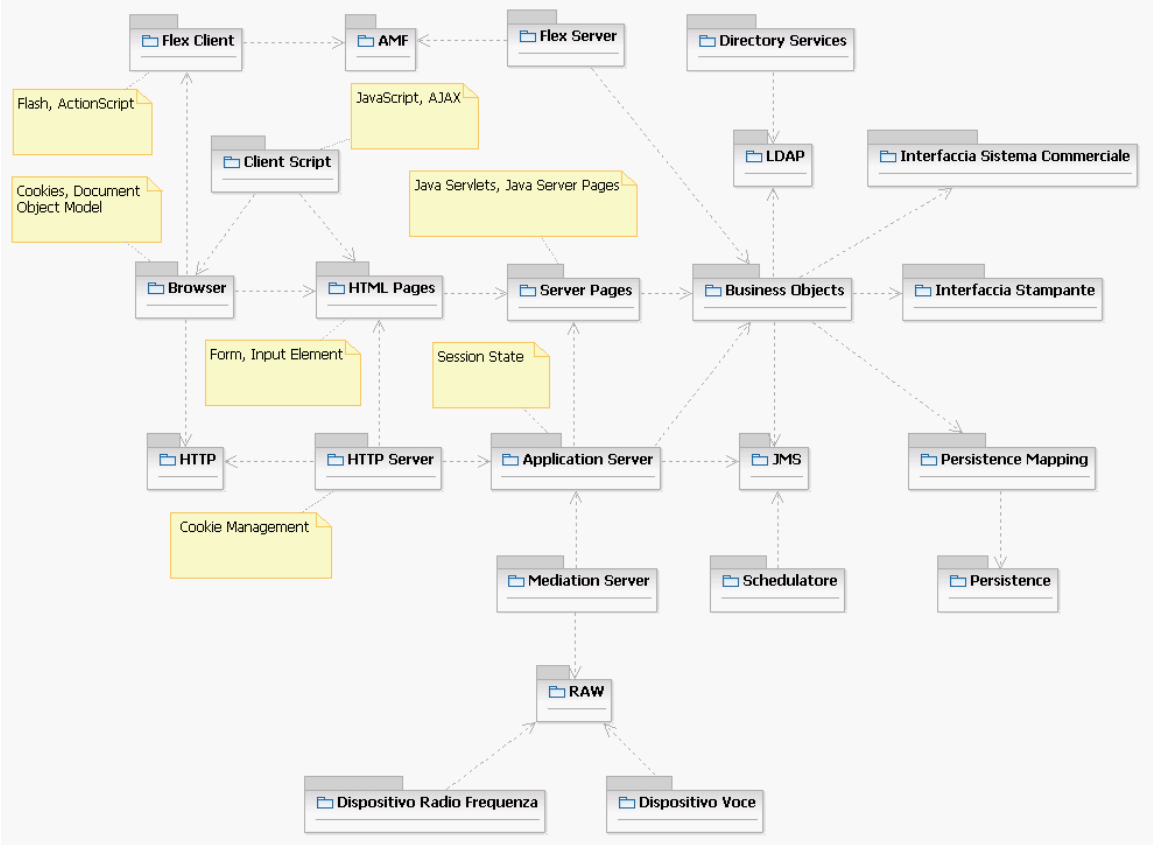

Figura 6: Le principali tecnologie utilizzate in OpenDedalo

I componenti di progettazione descritti in questa vista, evidenziano altre due scelte architetturali:

- il pattern Model View Controller (MVC) di separazione delle responsabilità dei componenti, in cui
- il Model si occupa di implementare la logica applicativa e di persistenza delle funzionalità
- il View si occupa di eseguire il rendering del risultato della funzionalità invocata
- il Controller si occupa di rispondere agli eventi provenienti dall'interfaccia utente, di invocare il Model opportuno e di scegliere la View che deve pubblicare la successiva mappa
- il meccanismo HTTP Session Object per la gestione dello stato conversazionale, in cui ogni richiesta di un dato utente durante la navigazione delle mappe è associata ad un contenitore, accessibile sia al componente di View che Controller, che può essere usato per mantenere informazioni trasversali alle funzionalità invocate, come il profilo dell'utente autenticato o lo stato delle informazioni inserite o selezionate durante un wizard.

3.1.3 Package Architetturalmente Significativi

I package rappresentati in questa vista, modellano tutti i dispositivi di interfaccia, protocolli, componenti, server, sistemi esterni, tecnologie e linguaggi di programmazione significativi per la progettazione dell'architettura.

Di questi, quelli che evidenziano le scelte principali sono:

- Browser, modella l'applicazione client usata dall'utente per interagire con il sistema
- HTML Pages, modella il modulo software implementato in HTML per la pubblicazione delle pagine web
- Client Script, modella il modulo software implementato in JavaScript per realizzare una comunicazione AJAX con il server

- Flex Client, modella il modulo software implementato in Flash ed ActionScript, eseguito all'interno di un plug-in del browser
- Server Pages (JSP), modella la tecnologia scelta per il rendering dinamico in HTML/JavaScript delle informazioni ritornate dai servizi invocati
- Business Objects, modella l'insieme di oggetti in cui risiede la logica applicativa, indipendentemente dalle modalità e canali con cui viene invocata
- Persistence Mapping, modella la configurazione di un modulo Object Relational Mapping (ORM) per la persistenza di oggetti su un database relazionale
- Schedulatore, modella il componente che, in base alla configurazione, invoca servizi di logica applicativa al raggiungimento di un dato istante temporale
- JMS (Java Message Service), modella lo standard di comunicazione con il sistema di gestione dei messaggi; tali code vengono usate per implementare logiche di esecuzione asincrona iniziate sia dall'utente che dallo schedulatore
- LDAP (Lightweight Directory Access Protocol), modella il protocollo di comunicazione con il Directory Server in cui è mantenuta l'anagrafica utenti ed il loro profilo base
- Mediation Server, modella il componente di mediazione che consente ai dispositivi a radio frequenza e vocali di interagire con il sistema

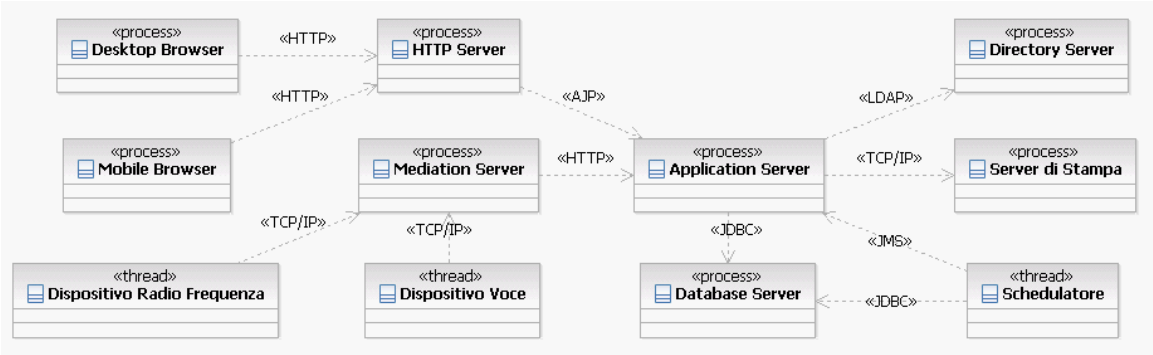

Figura 7: Comunicazioni all'interno di OpenDedalo

I principali processi o thread coinvolti nel sistema ed i package di cui ospitano l'esecuzione

sono:

- Desktop Browser, processo che ospita l'esecuzione dei package Browser, HTML Pages, Client Script e Flex Client; dialoga con il processo HTTP Server attraverso il protocollo HTTP
- Mobile Browser, processo che ospita l'esecuzione dei package Browser e HTML Pages; dialoga con il processo HTTP Server attraverso il protocollo HTTP
- Application Server, processo che ospita l'esecuzione dei package Server Pages, Business Objects e Persistence Mapping; i servizi di logica applicativa possono essere invocati attraverso i canali HTTP, AJP (Apache JServ Protocol) e JMS; il servizio di autenticazione invoca il Directory Server attraverso il protocollo LDAP, mentre quelli di persistenza invocano il Database Server attraverso connessioni JDBC
- Schedulatore, un thread per ogni attivazione che ospita l'esecuzione del package Schedulatore; dialoga con il processo Application Server per invocare servizi di logica applicativa; le attività schedulate possono accedere al Database Server attraverso connessioni JDBC
- Mediation Server, processo che ospita l'esecuzione del package Mediation Server; dialoga con i dispositivi a radio frequenza e vocali attraverso protocolli specifici.

3.1.1 Deployment View

Ogni processo o thread rappresentato nella Process View, trova corrispondenza in uno specifico nodo fisico che ne ospita l'esecuzione.

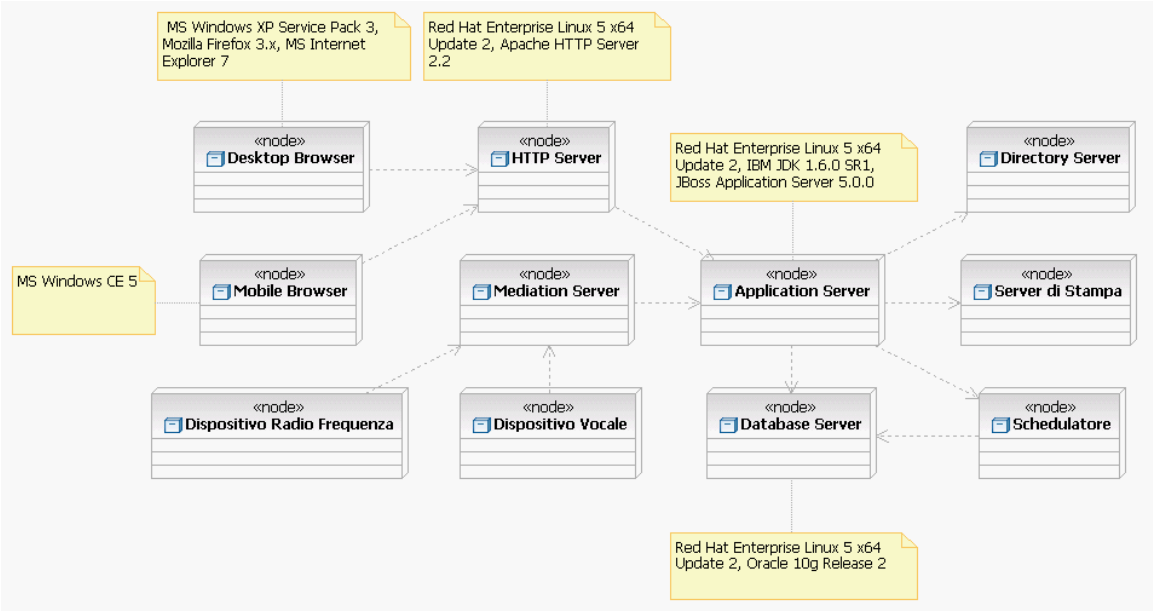

Figura 8: Implementazione dei moduli

3.1.2 Implementation View

Questa vista documenta quella parte dell'Implementation Model i cui package, layer e dipendenze da librerie esterne hanno maggior rilevanza architetturale. A questo livello di astrazione, ogni package modellato rappresenta un insieme di risorse (oggetti, documenti HTML, fogli di stile CSS, codice JavaScript, descrittori) che vengono rilasciate in un unico artefatto, come archivio di classi Java (JAR), archivio di risorse web (WAR), applicazione Flash (SWF).

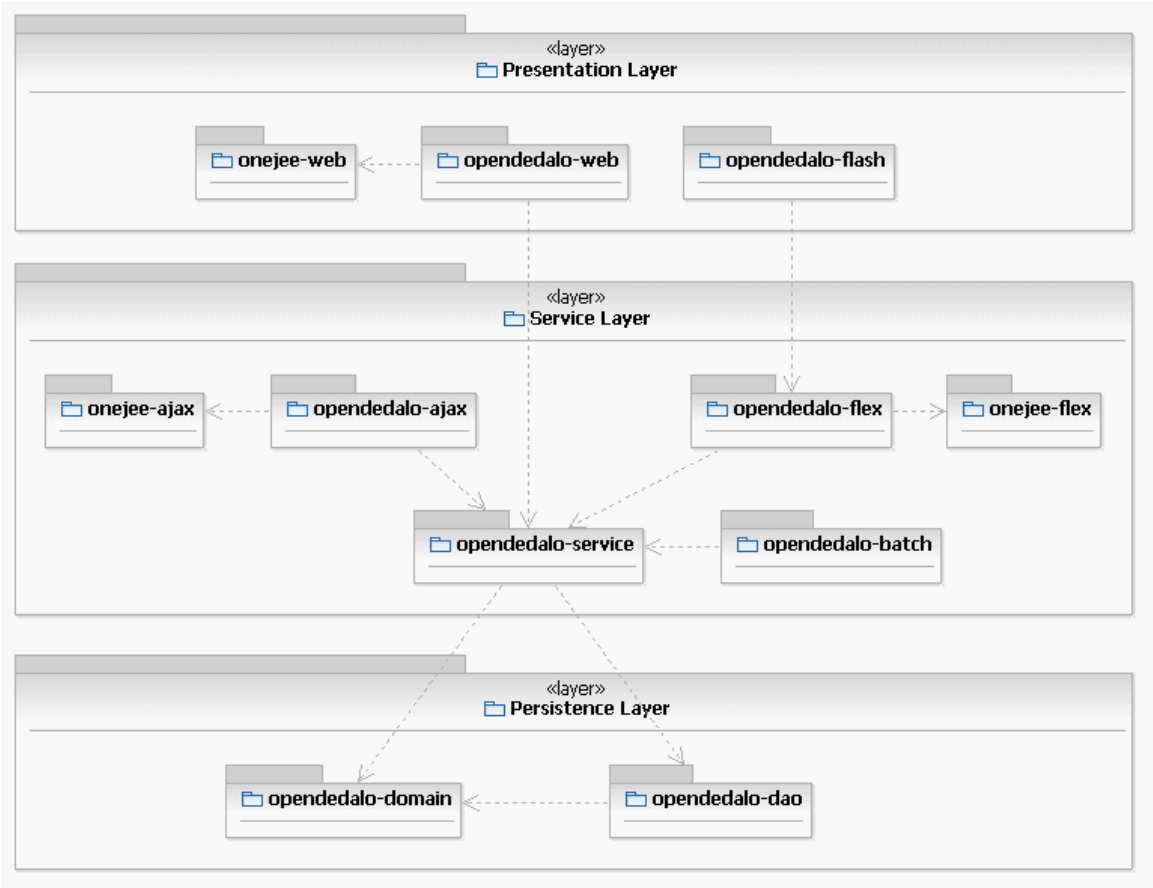

Figura 9: Layer di OpenDedalo

I package che completano ed integrano le librerie esterne usate a supporto dello sviluppo e che non implementano funzionalità specifiche di OpenDedalo (prefisso one*) costituiscono il Framework OneJEE.

Layers

A seconda delle responsabilità in carico ai diversi package, li possiamo suddividere nei seguenti tre layer:

- Presentation Layer, fanno parte di questo layer tutti i componenti coinvolti nell'attività di pubblicazione dei contenuti e di rendering dell'interfaccia utente
- Service Layer, fanno parte di questo layer tutti i componenti che implementano la logica di business dei servizi applicativi erogati
- Persistence Layer, fanno parte di questo layer tutti i componenti che implementano la persistenza delle informazioni (non solo su database relazionale)

Ogni package aggrega risorse omogenee per responsabilità, canale di accesso, modalità di attivazione:

- opendedalo-web, risorse che gestiscono l'interazione con il browser (Java Servlets, JavaScript) e che pubblicano i contenuti richiesti (Java Server Pages, Custom Tag, HTML, CSS)
- opendedalo-flash, risorse che gestiscono l'interazione con il plug-in Flash (MXML, ActionScript)
- opendedalo-service, oggetti che implementano le logiche business indipendentemente dal canale di accesso
- opendedalo-ajax, oggetti che mediano la richiesta di servizi sul canale AJAX verso gli oggetti del package opendedalo-service
- opendedalo-flex, oggetti che mediano la richiesta di servizi sul canale FLASH verso gli oggetti del package opendedalo-service
- opendedalo-batch, oggetti che, attivati su richiesta dello schedulatore, mediano verso gli oggetti del package opendedalo-service
- opendedalo-domain, oggetti persistenti che modellano le entità applicative (Domain Persistent Objects)
- opendedalo-dao, oggetti che eseguono le operazioni di ricerca e persistenza delle informazioni sia verso database relazionale (JDBC) che directory server (LDAP)

Lo sviluppo dei diversi package può contare sul supporto offerto da un insieme di librerie open-source, che ne semplificano la progettazione e realizzazione, oltre che costituire una guida all'adozione di pattern e best practices ormai consolidati in contesti applicativi enterprise.

- Spring Framework 2.5.6, container ad inversione di controllo [www.springsource.org/about]
- Spring Web MVC 2.5.6, implementazione del pattern MVC [www.springsource.org/about]
- Display Tag 1.2, custom tag per la gestione delle liste in pagine web [displaytag.sourceforge.net]

- Direct Web Remoting 2.0.5 (DWR), libreria per invocare servizi web in modalità AJAX [directwebremoting.org]
- BlazeDS 3.2.0, implementazione del protocollo si serializzazione AMF [opensource.adobe.com/wiki/display/blazeds/BlazeDS]
- Spring BlazeDS Integration 1.0.0, integrazione fra Spring Framework e BlazeDS [www.springsource.org/spring-flex]
- Quartz 1.6.4, schedulatore configurabile [www.opensymphony.com/quartz]
- Spring Security 2.0.4, supporto per l'autenticazione dell'utente (anche in configurazioni Single Sign On) e per l'autorizzazione all'esecuzione dei servizi [static.springframework.org/spring-security/site/index.html]
- Spring Integration 1.0.1, supporto all'integrazione con sistemi esterni [www.springsource.org/spring-integration]
- Ehcache 1.5.0, per il caching delle informazioni, con politiche di popolamento configurabili [ehcache.sourceforge.net]
- Spring Modules Cache 0.9, integrazione fra Spring Framework ed Ehcache [springmodules.dev.java.net]
- Hibernate Core 3.3.1, layer di Object Relational Mapping per il supporto alla persistenza di oggetti [www.hibernate.org]
- Spring LDAP 1.3.0, supporto per l'accesso ad un Directory Server attraverso il protocollo LDAP [www.springsource.org/ldap]
- AspectJ 1.6.2, supporto alla programmazione ad aspetti [www.eclipse.org/aspectj]
- JAMon 2.7 (Java Application Monitor), libreria per il monitoring dei tempi di esecuzione ed il livello di concorrenza di metodi di oggetti [jamonapi.sourceforge.net]
- Log4j 1.2.15, libreria per il logging su file system, database, code [logging.apache.org/log4j/1.2/index.html]
- Apache Commons, insieme di librerie di utilità varia (gestione collezioni, oggetti primitivi e non, logging, validazione, upload HTTP, mail, networking) [commons.apache.org]

4 Lo Svolgimento dello Stage

4.1 Scopo dello stage

Lo scopo del progetto è lo studio delle caratteristiche e funzionalità della libreria Spring BlazeDS Integration nell'ottica di un utilizzo nel progetto OpenDedalo. La libreria in questione fondamentalmente permette di integrare la gestione della comunicazione tra client e server all'interno del framework Spring. In questo modo la configurazione della libreria BlazeDS viene prodotta da un apposito file di contesto in modo analogo alle altre parti del sistema.

Per studiare in modo più diretto le librerie coinvolte, quindi il framework Spring, Flex e BlazeDS si è deciso di creare un progetto molto semplice che riproponesse la stessa struttura in package e moduli utilizzata da DEDALO.

Una volta prodotto e testato questo mini-progetto si è osservato che la libreria rispondeva alle necessità ricercate, in quanto permette con poche modifiche di configurare la libreria BlazeDS senza alterare il codice presente e senza introdurre regressioni o problemi per la sicurezza.

4.2 L'attività di Stage

La prima parte dell'attività di stage è stata relativa all'introduzione della libreria Spring BlazeDs Integration per gestire l'integrazione tra Spring e Flex. Questa è stata l'attività principale dello stage ed ha coperto due terzi della durata. Avendo già lavorato nello sviluppo di web application avevo già utilizzato alcuni strumenti come Eclipe o Turtoise SVN e conoscevo la struttura delle web application. Tuttavia, avendo sempre lavorato su progetti di piccole dimensioni, non mi era familiare l'architettura del progetto e la relativa suddivisione in sottoprogetti. Per fare un esempio, anche se avevo già utilizzato il framework Hibernate per accedere ai dati nel database non avevo mai utilizzato in maniera esplicita il pattern DAO.

Inoltre non avevo mai utilizzato il framework Spring o Flex. Quindi sono state necessarie un paio di settimane di studio. Dopo aver studiato le funzinoalità base di Spring ho approfondito i moduli relativi all'MVC e alla sicurezza. Per quanto riguarda BlazeDs è stato necessario studiare la documentazione ufficiale, alquanto scarna, integrando i punti mancanti con

informazioni provenienti da forum di sviluppatori.

4.3 Il progetto di test: SpringApp

Per avere un progetto dove eseguire i test senza avere il problema della complessità del codice applicativo è stato deciso di sviluppare un progetto di test chiamato SpringApp che avesse la stessa struttura di OpenDedalo, ma il minimo codice possibile.

L'applicazione in particolare non faceva altro che recuperare dei dati dal back end e visualizzarli a video. I dati venivano recuperati non da un database, ma inseriti direttamente nel codice. Per quanto riguarda la sicurezza si è decisi di usare solo l'autenticazione a livello di canale. In questo modo ho potuto studiare il comportamento di Spring BlazeDs nella gestione della sicurezza senza avere i problemi della gestione dell'infrastruttura. In OpenDedalo infatti viene utilizzato un servizio di LDAP al quale viene demandata la gestione dell'autenticazione e del recupero dei ruoli degli utenti.

Su questo progetto ho potuto provare le varie modalità con cui è possibile configurare la libreria di Spring BlazeDS, inoltre ho potuto provare le differenze tre la varie versioni della libreria. Nella documentazione ufficiale della libreria veniva consigliato di utilizzare l'ultima versione di Spring BlazeDS, dato che erano stati risolti alcuni problemi presenti con le versioni precedenti.

Per questo motivo si è deciso di utilizzare la versione 1.0.3 . Questo ha comportato un aggiornamento delle librerie di Spring alla versione 3.0.6 e delle librerie di Flex alla 3.2.0 .

4.4 BlazeDS più in dettaglio

Per entrare un po' nel dettaglio del funzionamento della libreria BlazeDs possiamo vedere come viene gestita una chiamata da un client nella figura seguente.

Il client esegue una chiamata usando un canale e la richiesta viene instradata a un endpoint sul server BlazeDS attraverso la servlet DispatcherServlet che funge da Controller per l'applicazione. Da questo endpoint la richiesta attraversa una catena di oggetti Java passando per il MessageBrokerServlet che funziona come dispatcher per le comunicazioni. Dal MessageBrokerServlet la richiesta viene inviata a un servizio e infine a una destinazione con

il relativo adapter. L'adapter esegue la richiesta o in locale o tramite un servizio JMS.

4.4.1 Endpoint

Gli Endpoint rappresentano il lato server del canale. Sono implementati come delle servlet, il che significa che l'I/O e le sessioni HTTP vengono gestiti da un J2EE container, in OpenDedalo da JBoss.

Gli endpoint vengono avviati dalla servlet MessageBrokerServlet, che viene configurata nel web.xml ed è il centro nevralgico della comunicazione tramite BlazeDs.

Dato che la comunicazione avviene attravero i canali viene definito una mappatura tra i canali sul lato client e gli endpoint sul server per garantire che venga utilizzato lo stesso formato dei messaggi per evitare problemi nella serializzazione e deserializzazione.

La configurazione degli endpoint è inserita nel file services-config.xml, tipicamente contenuto nella cartella WEB-INF/flex della web application.

4.4.2 Il MessageBrokerServlet

Il MessageBrokerServlet principalmente è responsabile dell'instradamento dei messaggi ai servizi. Quando un endpoint riceve una richiesta, estrae il messaggio e lo invia al MessageBrokerServlet che ispeziona la destinazione e quindi invia il messaggio al servizio opportuno. A questo livello vengono eseguiti i controlli di autenticazione e autorizzazione nel caso la destinazione lo richieda. Il MessageBroker viene configurato di default nel file services-config.xml nella directory WEB-INF/flex della web application.

4.4.3 Servizi e Destinazioni

BlazeDS mette a disposizione tre tipi di servizi e corrispondenti destinazioni:

• RemotingService and RemotingDestination

• HTPProxyService and HTTPProxyDestination

• MessageService and MessageDestination

I servizi sono il target dei messaggi originati dal client Flex. Le destinazioni si possono pensare come istanze di servizi configurate nel modo opportuno.

Ad esempio un componente RemoteObject viene usato nel client Flex per comunicare con il servizio RemotingService. Nel componente RemoteObject è necessario specificare l'id della destinazione che si riferisce all'oggetto Java esportato da BlazeDS

Il mapping tra componenti lato client Flex e servizi BlazeDs deve essere coerente, in particolare:

• Gli HTTPService e WebService comunicano attraverso HTTPProxyService/HTTPProxyDestination

• I RemoteObject comunicano con RemotingService/RemotingDestination

• Producer e Consumerc comunicano con MessageService/MessageDestination

I servizi e le destinazioni vengono configurate nel file services-config.xml, ma per seguire le best practice questo dovrebbe essere diviso in più file specifici, in particolare:

• I RemotingService configurati nel file remoting-config.xml

• Gli HTTPProxyService configurati nel file proxy-config.xml

• I MessageService configurati nel file messaging-config.xml

4.4.4 I file di configurazione

BlazeDS viene configurata nel file services-config.xml file posizionato, di default, nella directory WEB-INF/flex nell'applicazione web. Questo viene normalmente diviso in quattro file in base al tipo di servizio configurato.

Il file services-config.xml è il file di configuraione più importante. Contiene i vincoli relativi

alla sicurezza, la definizione dei canali e le impostazioni per eseguire il logging specifiche per ogni servizio e utente.

Il file remoting-config.xml contiene la configurazione delle destinazioni utilizzate per lavorare con gli oggetti remoti.

Il file proxy-config.xml definisce le destinazioni per i Proxy Service per lavorare con i servizi web e HTTP.

Il file messaging-config.xml definisce le destinazioni per eseguire la comunicazione in modalità publish-subscribe.

4.4.5 Spring BlazeDS

A differenza di quanto era necessario fare con la libreria di Adobe, per utilizzare Spring BlazeDS è sufficiente configurare il MessageBroker nel web.xml come una normale servlet gestita da Spring. In questo modo tutti i messagi provenienti dal client Flex verranno indirizzati dal DispatcherServlet di Spring al MessageBroker.

Dal punto di vista pratico si aggiunge nei vari file di contesto Spring un namespace flex che permette di gestire la configurazione di BlazeDS con pochi tag.

Per fare un esempio della semplicità di configurazione di Spring BlazeDS rispetto alla versione di Adobe possiamo osservare che per configurare il MessageBroker avremmo dovuto almeno configurare un MessageBrokerFactoryBean, un MessageBrokerHandlerAdapter e un HandlerMapping per instradare le richiesta, mentre con la libreria di Spring questo viene gestito con un semplice tag che definisce il MessageBroker, cio con qualcosa di simile a <flex:message-broker/> all'interno di un file di contesto.

Ovviamente le strutture che non configuriamo esplicitamente vengono ancora utilizzate. In generale le impostazioni utilizzate di default sono sufficienti per la maggior parte delle applicazioni e quindi non è necessario sovrascriverle. Anche la sovrascrittura di queste impostazioni è stata semplificata, ad esempio per definire un percorso diverso del file services-config.xml è sufficiente modificare

nel modo seguente l'elemento message-broker

```
<flex:message-broker services-config-path="classpath*:services-config.xml"/>
```

mentre utilizzando la sintassi di Spring sarebbe necessario un codice di questo tipo:

```
<bean id="_messageBroker"
 class="org.springframework.flex.core.MessageBrokerFactoryBean" >
      <property name="servicesConfigPath"
        value="classpath*:services-config.xml" />
</bean>
```

4.4.6 Traduzione delle eccezioni

Uno dei problemi che erano sorti utilizzando la libreria BlazeDS di Adobe è che, quando veniva sollevata un'eccezioni nel server, il codice che veniva inviato al client era troppo generico per esporre all'utente una spiegazione dettagliata. Questo comportava che quando il lato server andava in errore venisse sollevato un errore nel lato client con il messaggio generico “Server.Processing ”

Per risolvere questo problema è possibile utilizzare dei traduttori di eccezioni che convertono le eccezioni nell'appropriato oggetto Flex MessageException.

Dal punto di vista pratico è sufficiente configurare nel MessageBroker la proprietà message-interceptor con un opportuno bean come nell'esempio di codice seguente:

```
<bean id="myMessageInterceptor"
     class="com.foo.app.MyMessageInterceptor"/>

<flex:message-broker>
     <flex:message-interceptor ref="myMessageInterceptor"/>
</flex:message-broker>
```

La classe che intercetta i messaggi di errore deve implementare l'interfaccia org.springframework.flex.core.MessageInterceptor .

Questa interfaccia contiene due soli metodi. Il metodo handles(Class clazz) viene invocato per decidere se l'eccezione sollevata deve essere gestita, se il metodo ritorna un valore positivo viene invocato il metodo translate(Throwable trw) che riceve in input l'eccezione sollevata e restituisce un MessageException valorizzato nel modo opportuno. Questo sarà poi disponibile all'interno di un evento FaultEvent e dovrà essere gestito in modo opportuno all'interno del client.

4.5 La soluzione utilizzata

Per utilizzare il MessageBroker all'interno della struttura MVC di Spring è stato necessario aggiungere una servlet che svolga il ruolo di Controller per i messaggi provenienti dal client, in questo caso caratterizzati dall'url messagebroker.

Per questo è stata aggiunta una servlet chiamata flexMessageBroker e configurata tramite il file flex-servlet.xml . Nel seguito il codice necessario nel web.xml

```
<servlet>
    <servlet-name>flexMessageBroker</servlet-name>
    <servletclass>
                org.springframework.web.servlet.DispatcherServlet
    </servlet-class>
    <init-param>
         <param-name>contextConfigLocation</param-name>
         <param-value>/WEB-INF/flex-servlet.xml</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
</servlet>

<servlet-mapping>
    <servlet-name>flexMessageBroker</servlet-name>
    <url-pattern>/messagebroker/*</url-pattern>
```

```
    </servlet-mapping>
```

Il file flex-servlet.xml contiene la configurazione del MessageBroker ed è riportato in seguito.

```
    <flex:message-broker >
        <flex:exception-translator ref="flexExceptionTranslator" />
        <flex:message-service default-channels="my-amf,
            my-secure-amf,my-polling-amf" />
        <flex:secured>
            <flex:secured-channel channel="my-amf"
                access="ROLE_USER" />
            <flex:secured-channel channel="my-secure-amf"
                access="ROLE_USER" />
            <flex:secured-channel channel="my-polling-amf"
                access="ROLE_USER" />
            <flex:secured-endpoint-path access="ROLE_USER"
                pattern="**/messagebroker/*"/>
        </flex:secured>
    </flex:message-broker>
```

Possiamo osservare che questo codice è molto semplice e praticamente autoespilcativo. Il MessageBroker è un normale bean istanziato come singleton anonimo. Gli viene passato il riferimento alla classe che traduce i codice delle eccezioni e vengono definiti i canali di default. I canali sono un semplice canale in AMF, un canale AMF con vincoli di sicurezza e un canale AMF con il polling abilitato. Questi vengono definiti come secured, ovvero per accedervi l'utente deve essere autenticato come “ROLE_USER”. Questo serve a integrare nell'uso dei canali Flex le impostazioni di sicurezza utilizzate nel resto dell'applicazione e definite in un apposito file di contesto.

All'interno dello stesso file flex-servlet.xml sono anche definiti gli oggetti remoti esportati nel client. Per fare questo è sufficiente definire le destination dei bean opportuni, che vengono configurati in un file di contesto dedicato.

Nel seguito un esempio di servizio esportato come oggetto remoto.

```
<flex:remoting-destination destination-id="precaricoFlexService"
    ref="precaricoFlexService" />
```

Un'alternativa a elencare nel *flex-servlet.xml* le destinazioni esportate è di utilizzare le annotazioni nelle classi che implementano il servizio, tuttavia si è previsto questo approccio considerando che l'alternativa comportava modificare manualmente tutte le classi. Inoltre è in linea con la filosofia di Spring mantenere un unico file xml di configurazione.

L'ultimo elemento necessario per configurare completamente la comunicazione con la libreria *Spring BlazeDS Integration* è il file *services-config.xml*. In questo file, posizionato di default nella cartella *WEB-INF/flex*, vengono definiti i canali e gli *endpoint* con le impostazioni relative ai server.

4.6 La prima impostazione del Front end di OpenDedalo

Durante lo sviluppo del progetto *OpenDedalo* la maggior parte delle forze sono state impiegate nello sviluppo del model dell'applicazione, trascurando in prima battuta il presentation layer. Per questo motivo è stata utilizzata un architettura molto semplice basata su una tripla di oggetti: la pagina, il modello e il modulo.

Il modello è la classe fondamentale ed è qualcosa di simile ad un Java bean. Contiene tutti gli elementi che devono essere visualizzati nella pagina, sia tipi semplici, sia altri oggetti.

Il modulo è un file actionscript che contiene tutte le funzioni che vengono utilizzate nella pagina.

Il collegamento tra questi tre oggetti avviene nella pagina mxml. In questa viene importato il modulo come file di script e il modello viene definito *Bindable*. L'uso di questo meta-tag del linguaggio Flex implica che per ogni proprietà dell'oggetto venga istanziato un listener che, alla modifica della proprietà, vada ad aggiornare tutti i punti della pagina che la utilizzano e a notificare a tutti i listener relativi l'evento. Nonostante sia uno strumento molto potente, il tag *Bindable* è in realtà adatto solo ad applicazioni molto piccole. Infatti vengono istanziati listener per tutte le proprietà, anche per quelle che non vengono utilizzate nella pagine. In secondo luogo l'associazione tra proprietà osservata e listener è fatto tramite una strong

reference che quindi non viene eliminata dal garbage collector finchè l'applicazione intera non viene terminata. La conseguenza di questa impostazione è che il consumo di memoria dell'applicazione cresce costantemente durante la navigazione dell'utente e spesso porta a un crash del Flash Player (con un messaggio di errore generico). L'utilizzo della memoria è infatti un punto critico del Flash Player in quanto è praticamente una macchina virtuale che viene avviata all'interno del browser.

Per risolvere questo problema si è deciso di sviluppare una nuova architettura del front-end eliminando l'uso dei tag Bindable basata sui pattern Observer e Mediator.

4.7 I Pettern Utilizzati nella nuova architettura

4.7.1 Il pattern Mediator

Il design pattern mediator è un pattern comportamentale basato sugli oggetti.

Il problema che si vuole risolvere con questo pattern è il mantenimento efficiente di relazioni molti a molti tra oggetti. L'obiettivo è vuole mantenere l'accoppiamento tra questi oggetti quanto più lasco possibile, in maniera da evitare il problema del codice “tipo spaghetti” in cui per usare un oggetto si usano anche tutte le relazioni in cui è coinvolto a cascata. Si vuole mantenere la modularità e riusabilità del singolo componente.

Uno scenario in cui questo problema si presenta è il mantenimento dei ruoli, ad esempio nei sistemi di tipo Unix. Un utente spesso fa parte di più di un gruppo e se mantenessimo le relazioni tra utenti e gruppi dovremmo aggiornare l'intero sistema ogni volta che si presenta una variazione. Lo stesso problema si presenta mutatis mutandis nelle web application in cui devono essere gestiti i ruoli per garantire l'accesso alle risorse dell'applicazione solo agli utenti che ne hanno i permessi.

La soluzione proposta usando questo pattern è di introdurre un ulteriore livello di indirection trasformando le relazioni molti a molti in oggetti. In questo modo viene introdotto un oggetto Mediatore che si comporta come hub di comunicazione controllando e coordinando tutte le comunicazioni tra i vari oggetti. In questo modo è possibile inserire nel Mediatore eventuali logiche che controllano le relazioni.

Nel seguito viene mostrata la struttura della soluzione. In questo esempio estremamente semplice vediamo che il *Mediator* si occupa di gestire le relazioni tra i due oggetti *Producer* e *Consumer*, che infatti non comunicano direttamente tra loro.

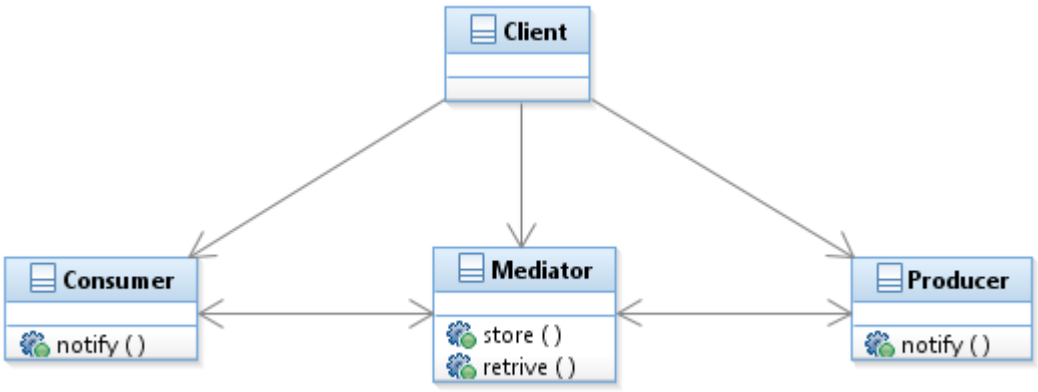

Figura 9: Diagramma del pattern Mediator

Per essere un po' più specifici, il pattern viene implementato utilizzando due soli tipi di elementi: il *Mediator* e il *Colleague*. In una posibile implementazione questi estenderebbero due interfacce come si vede nella figura seguente.

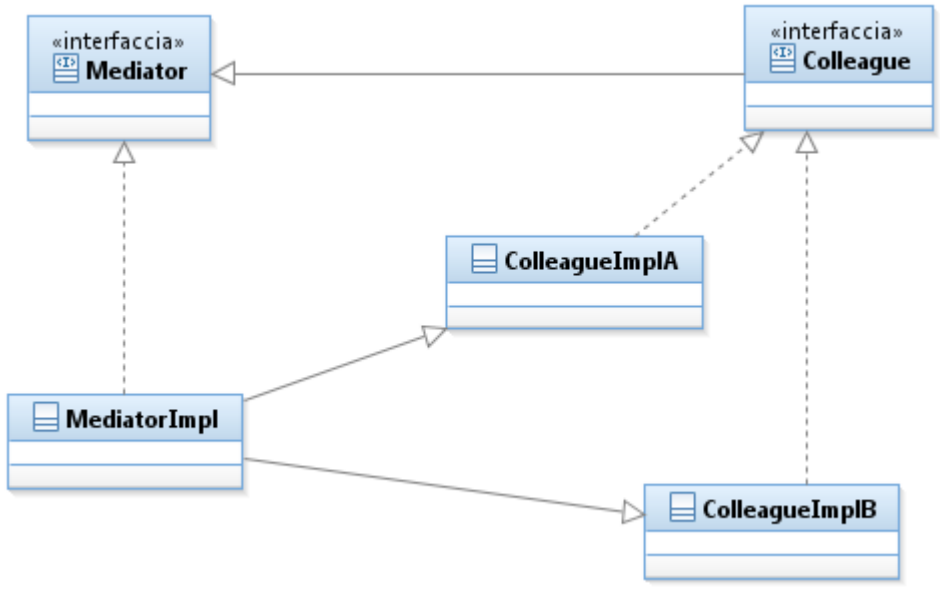

Figura 10: Implementazione del pattern Mediator

Gli attori base di quest'implementazione sono i seguenti:

•*Mediator*: definisce una interfaccia per comunicare con i *Colleague*

•*MediatorImpl*: implemente l'interfaccia *Mediator,* mantiene la lista dei colleghi e implementa lo scambio di messaggi tra di loro

•*Colleague*: definisce l'interfaccia dei colleghi

•*ColleagueImpl*: implementa il singolo collega e le modalità di comunicazione con il *Mediator*

I vantaggi che questo pattern presenta sono:

•Eliminare le connessioni tra colleghi: questi dialogano tra di loro in modo indiretto passando per il *Mediator* e questo facilita la gestione delle comunicazioni

•Semplificare le connessioni: il *Mediator* consente di ridurre le connessioni dei Colleghi da many-to-many a one-to-many

•Controllo centralizzato: il controllo delle comunicazioni è centralizzato e questo consente di avere una visione complessiva del sistema ed una gestione più efficente delle modifiche

•Single Point of Failure: nel caso di malfunzionamento del *Mediator* l'intero sistema sarà coinvolto ed in caso di fermo del *Mediator*, i Colleghi resteranno isolati, eventuali problemi isolati ai Colleghi non pregiudicheranno le connessioni del resto del sistema.

4.7.2 Il Pattern Observer

Il pattern *Observer* è un desgin pattern comportamentale progettato per gestire la comunicazione tra più oggetti.

Consideriamo la situazione in cui lo stato di uno o più oggetti, detti osservatori, debba essere essere mantenuto costantemente aggiornato con quello di un dato oggetto, detto soggetto, che varia in risposta a azioni non derivate dagli osservatori.

Si vuole creare un meccanismo per cui il soggetto informa gli osservatori ogni volta che il suo stato subisce una variazione, in modo che questi possano reagire nel modo opportuno.

Il paradigma corrisponde al modello *Publish and Subscribe*: i sottoscrittori si registrano presso un *Publisher* e quest'ultimo li informa ogni volta che ci sono nuove notizie (del genere sottoscritto).

La figura seguente è una schematizzazione del pattern a livello concettuale, dove un soggetto comunica le proprie modifiche a vari osservatori.

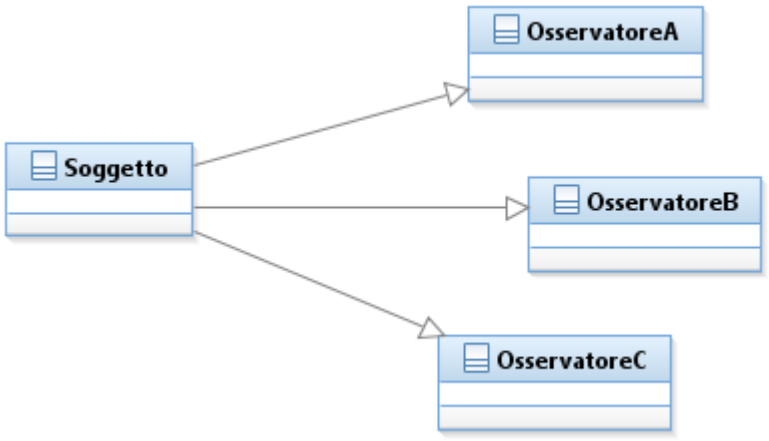

Figura 11: Il pattern Observer

Il diagramma UML del design pattern è nella figura seguente.

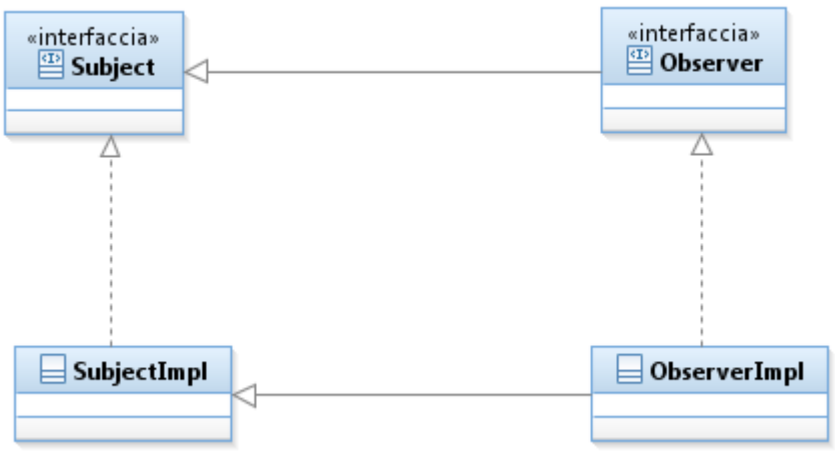

Figura 12: Schema UML del pattern Observer

Subject è l'interfaccia implementata dal soggetto concreto, mentre *Observer* è l'iterfaccia implementata dagli osservatori concreti. Il soggetto tiene i riferimenti agli osservatori per poterli avvisare, mentre gli osservatori concreti tengono il riferimento al soggetto per poterne leggere lo stato.

Più in dettaglio specificatamente la classe soggetto deve prevedere i seguenti metodi:

- *subscribe()* chiamato da un osservatore che intende registarsi;
- *unsubscribe()* chiamato da un osservatore che intende deregistrarsi.

Inoltre il soggetto deve prevedere un modo per informare gli osservatori di un avvenuto cambiamento di stato.

4.7.3 L'uso dei pattern nella soluzione proposta

I pattern *Mediator* e *Observer* sono stati utilizzati insieme per venire incontro alle esigenze dell'applicazione.

Per presentare in breve l'architettura proposta è utile seguire il seguente schema, dove viene mostrata in modo molto semplificato. Concettualmente quello che si è fatto è stato usare un

oggetto centralizzato, l'*ObserverManager* per gestire tutti gli aggiornamenti degli oggetti usati nella pagine e un oggetto *PresentationModel* associato a ogni pagina per per gestirne tutte le operazioni.

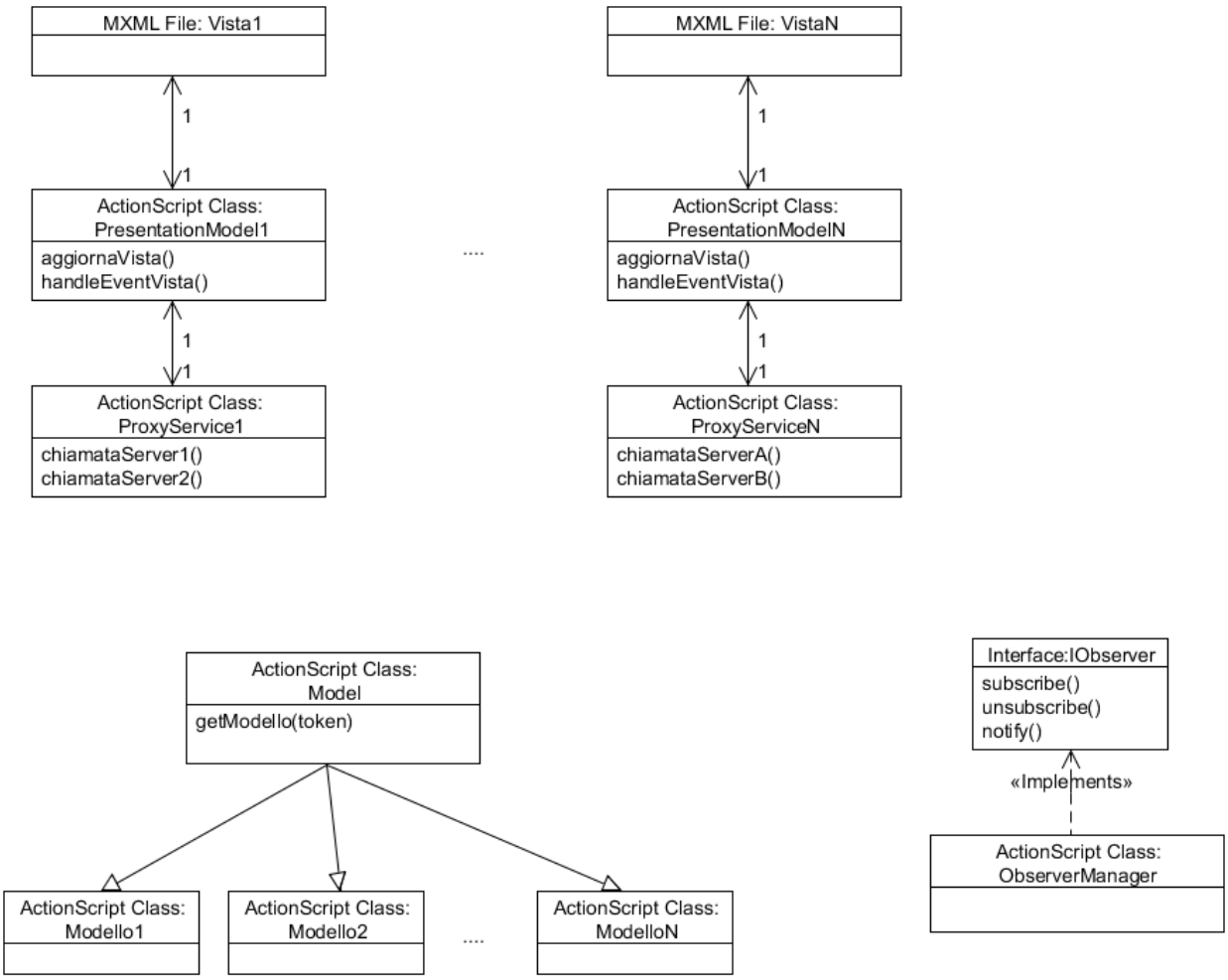

Figura 13: Schema della soluzione proposta

I primi oggetti che vengono rappresentati sono le pagine dell'applicazione. Ogni pagina corrisponde a un file mxml, ovvero al una pagina XHTML prodotta con il linguaggio di marcatori di Macromedia. A ogni pagina viene associato un *PresentationModel*. Questo è l'oggetto centrale dal punto di vista logico, infatti concentra al suo interno tutta la gestione del flusso di informazioni della pagina. Ogni pagina ha associato un *PresentationModel* sviluppato ad hoc. Questo per eseguire le chiamata al lato server dell'applicazione contiene una associazione a un oggetto che funziona da proxy.

Vengono usate delle relazioni uno-a-uno tra gli oggetti utilizzati in ogni pagina. In questo modo la pagina, cioè la Vista, i metodi presenti nel PresentationModel e le chiamate al server eseguite dai ProxyService vengono isolati per area funzionale e per ogni istanza di pagina. Ogni pagina contiene una tripla di questo tipo.

Nella prima impostazione del front-end il modello veniva collegato alla pagina usando il meta-tag *Bindable* che creava una strong reference con i relativi problemi per la gestione del *garbage collector*. Nella soluzione proposta il modello viene incluso nella tripla e recuperato tramite una classe che funge da *Façade*. In questo modo non è più necessario inserire nella pagina un link al tipo di *Model* corrispondente, ma solamente una chiamata alla *Façade,* che si preoccupa di fornire un'istanza del *Model* opportuna.

Quindi ogni area funzionale contiene una quadrupla di questi quattro elementi. Inoltre nel caso si aprissero più pagine dello stesso tipo, verrebbero prodotte più istanze delle pagine, ognuna gestita correttamente in isolamento rispetto alle altre.

La differenza rispetto all'impostazione precedente è che tutta la gestione delle logiche viene demandata al *PresentationModel* e tutti gli aggiornamenti all'*ObserverManager.*

Il *PresentationModel* non fa altro che inizializzare degli *handler* per tutti gli eventi che sono rilevanti per la pagina e registrarsi come osservatore per tutti gli oggetti che possono essere aggioranti.

In questo modo, quando un modello viene modificato da una qualsiasi fonte, l'*ObserverManager* richiama il *PresentationModel* opportuno e questo esegue gli aggiornamenti necessari nella pagina o in genere le azioni adatte.

Nel seguito la soluzione proposta viene analizzata più in dettaglio.

4.8 La Nuova architettura

La nuova architettura del front-end di OpenDedalo si basa sugli stessi oggetti usati in precedenze.

- Una classe *actionscript, Modello,* che contiene i dati che devono essere presentati in pagina e è quindi relativa a un area funzionale. Concettualmente è simile a un Java Bean

- Un file *mxml* che contiene la struttura grafica della pagina (in modo analogo a quanto accade con una jsp i file mxml vengono compilati come una classe actionscript generando un file swt che poi viene eseguito dal plug-in flash player del browser).
- Una classe *PresentationModel* che lega le due classi precedenti e ne gestisce le interazioni.

  L'unica entità che viene aggiunta alla struttura precedente è una classe *proxy* che gestisce le chiamate al *back-end* Java.

Ogni area funzionale del progetto ha questa quadrupla di entità associate.

Per gestire in modo più efficiente il recupero del modello evitando di ricaricare più volte lo stesso modulo (e quindi di andare incontro a problemi di occupazione di memoria del Flash player) viene utilizzata una classe *Sessione* che è un singleton e che ha il compito di recuperare come *Façade* le istanze del *Modello* associate a una della vista. Inoltre viene utilizzata una classe *ObserverManager* che implementa il pattern di design *Observer* e gestisce le comunicazioni tra i vari elementi.

Per caricare un modulo nell'applicazione viene utilizzata un'istanza della classe *mx.modules.ModuleLoader*[6] che viene istanziata dall'applicazione. Nel serguito viene mostrato un esempio di funzione di caricamento del modulo *InterrogazioneMovimenti* che viene attivata da un evento di tipo *CreationComplete*.

```
public function inizializzaTestMovimentiMagazzino(flexEvent:Event):void {
    var moduleLoader: ModuleLoader= new ModuleLoader( );
    moduleLoader.applicationDomain = ApplicationDomain.currentDomain;
    pannelloPrincipale.addChild(moduleLoader);
```

6 Il componente ModuleLoader viene utilizzato tipicamente per caricare all'interno dell'applicazione le pagine mxml. La sua caratteristica principale è che gli oggetti che vengono caricati devono implementare l'interfaccia IflexModuleFactory. Questo significa che è possibile creare istanze multiple della classe filgio caricata.

```
        moduleLoader.url = Sessione.baseModulePackage +
            'ModuloInterrogazioneMovimenti.swf';
        ObserverManager.getInstance().register(moduleLoader,this);
    }
```

Il *ModuleLoader* viene istanziato come figlio del *pannelloPrincipale*, ovvero dell'elemento grafico della pagina nel quale dovrà essere caricato il modulo *InterrogazioneMovimenti*.

Per attivare il caricamento del modulo è sufficiente valorizzare la proprietà url con il percorso della classe corrispondente al modulo. Il Flash Player provvede di conseguenza a caricare la classe selezionata.

La classe *ModuloInterrogazioneMovimenti* in realtà è molto semplice. Infatti estende la classe *ModuloBase* specificandone i riferimenti alla vista e al *PresentationModel*. Il costruttore del *ModuloBase* crea un *eventHandler* che viene chiamato al momento dell'istanziazione della classe e che non fa altro che associare al *PresentationModel* specifico il modulo stesso.

Dopo che il modulo e il *PresentationModel* sono stati associati quest'ultimo viene inizializzato chiamando delle funzioni che vengono ereditate del *PresentationModel* specifico chr estende la classe *PresentationModelBase*.

In particolare la funzione attivata alla creazione del modulo è la seguente.

```
    private final function creationCompleteHandler(event: FlexEvent):void{
        presentationModelBase = getPresentationModel();
        presentationModelBase.initPresentationModel();
        presentationModelBase.startModule();
    }
```

Dato che Flex non prevede un costrutto sintattico corrispondente alla classe astratta di Java

per aggiungere dei metodo che devono essere obbligatoriamente sovrascritti dalla sottoclasse, si usa sollevare un eccezione all'interno del metodo della superclasse.

Quindi ad esempio il metodo *getPresentationModel()*, che deve restituire il *PresentationModel* specifico è così definito nella classe *ModuloBase*:

```
public function getPresentationModel():IPresentationModel{
    throw(new Error('fare l'override di
        IModuloBase.getPresentationModel'));
}
```

Il metodo della classe *PresentationModel* *initPresentationModel*() inizializza nel *PresentationModel* il *moduleLoader*, il *modulo* e il *proxy* e si registra *nell'ObserverManager*.

Il *moduleLoader* viene utilizzato come token per recuperare l'istanza corretta dell'oggetto osservato.

```
public final function initPresentationModel():void{
   _moduleLoader = getModuleLoader();
   ObserverManager.getInstance().register(_moduleLoader,this);
   _modelloBase = getModello() as ModelloBase;
   _baseProxyService = getModuleProxyService() as ProxyServiceBase;
}
```

La funzione *getModuleProxyService()* viene sovrascritta nella classe *PresentationModel* per restituire il *proxy* specifico del modulo. La funzione *startModule()* specifica di ogni modulo serve a inizializzare nel modo opportuno la vista della pagina, associando agli elementi gli *eventHandler* previsti.

Nel seguito viene mostrato un diagramma del flusso di attivazione di un modulo.

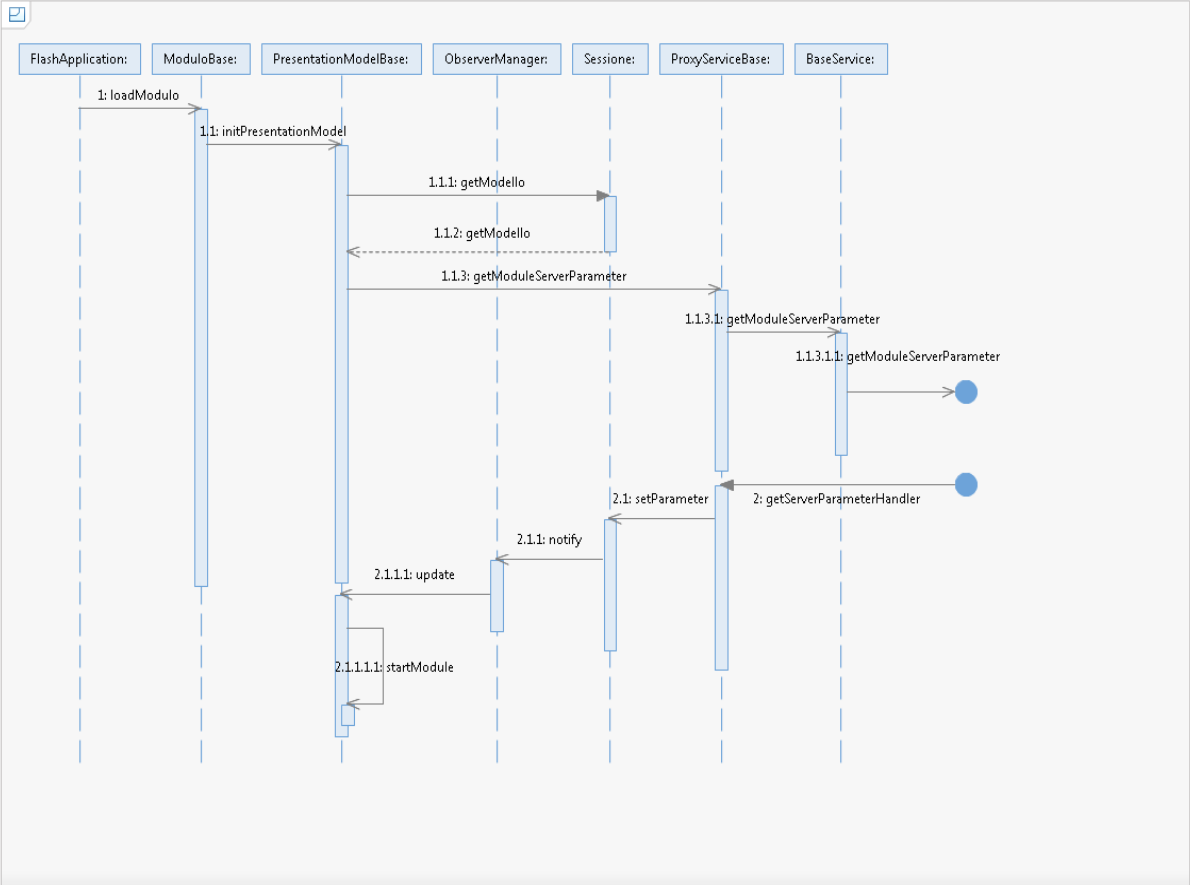

Figura 14: Diagramma di attivazione di un modulo

Dal diagramma precedente vediamo che a seguito di un'azione da parte dell'applicazione, ad esempio avviata dall'utente, viene inizializzato un *moduloBase*, che a sua volta inizalizza un *PresentationModelBase*; questo recupera l'istanza opportuna del *Modello* dalla *Sessione* e tramite il proxy. L'aspetto interessante che viene evidenziato dal diagramma è che la risposta del server viene memorizzata nel modello e che questo tramite l'ObserverManager notifica al PresentationModel che i dati sono stati aggiornati. Di conseguenza il PresentationModel completa l'inizializzazione della pagina con i parametri ottenuti dal server.

L'aspetto che si vuole sottolineare con questo diagramma è che il ciclo di attivazione e inizializzazione di una pagine viene gestito dalle classi base che vengono poi estese dalle classi specifiche della funzionalità che si sta usando. Questo ha permesso di applicare la nuova architettura al codice che era già stato sviluppato senza produrre delle modifiche sostanziali, semplicemente facendo estendere alle classi già esistente le classi di base previste

dalla nuova impostazione ed eliminando il file actionscript che veniva importato in ogni pagina.

Il diagramma illustra l'attivazione e inizializzazione di una pagina, ma la logica che gestisce le operazioni in OpenDedalo è praticamente la stessa. É stato usato come esempio l'inizializzazione di un modulo perchè tutte le chiamate vengono eseguite dalle classi base, mentre per le logiche applicative, pur restando la stessa struttura, lo schema si complicherebbe aggiungendo a ogni classe base la sua versione specifica dell'area funzionale.

4.8.1 La Sessione

I modelli vengono recuperati da una classe che svolge il ruolo di *Façade* e gestisce l'inizializzazione e il recupero delle istanze dei modelli.

Per funzionare da *Façede* la *Sessione* deve essere una classe singleton. Dato che Flex non prevede un costrutto sintattico per definire una classe come singleton viene usato un workaround passando al costruttore una istanza di una classe privata accessibile solo dall'interno della classe che si vuole rendere singleton.

Ad esempio nel codice seguente il costruttore riceve come parametro una istanza della classe *SingletonEnforcer* che è una classe privata che viene istanziata in un solo punto del codice ovvero dal metodo statico *getInstance()*.

```
public static function getInstance():Sessione{
    if ( !_instance )
        _instance = new Sessione( new SingletonEnforcer() );
     return _instance;
}
```

```
public function Sessione(se:SingletonEnforcer){
    if ( !se )
        throw( new Error( 'Usare Sessione.getInstance()' ) );
    mappaDelleMappeDeiModelli = new Object();
}
```

I modelli vengono gestiti tramite una struttura dati semplice: un array associativo che usa come chiave una stringa con il nome del modello e come valore un array associativo che usa come chiave l'istanza del *ModuleLoader* e come valore l'istanza del modello corrispondente. In questo modo con il nome della classe del modello si recupera la lista delle istanze attualmente in uso nell'applicazione e con il M*oduleLoader* l'istanza specifica che si cerca.

```
public function getModello(nomeModello:String,observable:Object):IModello
{   if(observable == null || nomeModello == null) return null;
    var mappaModello : Object = mappaDelleMappeDeiModelli[nomeModello];
    if(mappaModello == null){
        mappaModello = new Object();
        mappaDelleMappeDeiModelli[nomeModello] = mappaModello;
    }
    var modello:IModello =  mappaModello[observable];
    if(modello == null){
        var classeModelDefinition:Class = getDefinitionByName(
            IMODELLO_PACKAGE_BASE + '.' + nomeModello);
        var myClassFactory:ClassFactory =
```

```
                new ClassFactory(classeModelDefinition);
            modello = myClassFactory.newInstance( );
            modello.observable = observable;
            mappaModello[observable] = modello;
        }
        return modello;
    }
```

I parametri passati sono una stringa che rappresenta il nome completo della classe actionscript che si sta cercando e un oggetto che rappresenta l'istanza del *ModuleLoader* che ha caricato la vista associata al modello che si cerca. La funzione gestisce anche l'inizializzazione dei modelli e delle strutture dati per gestire il caso in cui vengono cercati per la prima volta i modelli.

4.8.2 I Proxy

I *ProxyService* vengono utilizzati per gestire le chiamata al back-end, ovvero le chiamata agli oggetti remoti Java esportati tramite *BlazeDs*. Ogni tripla *Modello*, vista, *PresentationModel*, viene associata a una istanza specifica di *ProxyService* che viene ottenuta nell'inizializzazione del *PresentationModel* da un metodo sovrascritto della classe base così come avviene per gli altri oggetti specifici dell'area applicativa.

Il *ProxyService* estende la classe *ProxyServiceBase* dalla quale eredita le chiamate comuni a tutte le pagine dell'applicazione in cui vengono recuperati dal back-end dei parametri relativi alla formattazione della pagina.

Al momento dell'inizializzazione del *ProxyService* gli vengono passati i due oggetti necessari per la sua esecuzione, ovvero, il *Modello* che contiene i parametri con i quali eseguire le chiamate al *remote object* e il *remote object* stesso. Infatti, quando viene invocato un metodo

del *proxy*, questo non fa altro che prendere dal modello i parametri necessari ed eseguire la chiamata. Quando viene creato l'oggetto che rappresenta il metodo Java che viene invocato, gli vengono associati due opportuni *handler* che dopo averlo eseguito settano sul modello il valore di ritorno della chiamata. Nei vari metodi *setter* del modello sono incluse le chiamate *all'ObserverManager* che provvedea notificare al corretto *PresentationModel* che c'è stata una modifica nel *Modello* che deve essere gestita, tipicamente mostrata in pagina.

4.8.3 I Service

La classe *Service* gestisce la creazione dei *remote object* esporati del back-end Java. I *remote object* chiamati nell'applicazione servizi sono classi java che vengono esportate da BlazeDs definendo un canale formato ai due lati da *destination* e *end-point*. L'id della *destination* è praticamente il nome che viene utilizzato dal lato flex per creare l'oggetto remoto. Le varie classi *Service* estendono una classe base che contiene le chiamate ai metodi che vengono utilizzati da ogni pagina dell'applicazione, ad esempio quelli che recuperano i parametri della formattazione della pagina.

Oltre alla creazione del *remote object* la classe *Service* è responsabile della creazione dell'*Operation*, ovvero dell'oggetto flex che rappresenta il metodo che viene invocato sulla classe Java. Dato che i parametri vengono recuperati dal modello e quindi devono essere gestiti del *ProxyService* la classe *Service* si limita a creare e restituire l'*Operation* associandovi i due handler che andranno a gestire le chiamata con esito negativo e positivo. Un'altra funzione che viene recuperata dalla classe base è appunto l'inizializzazione dell'*Operation*. Questa non fa altro che aggiungere i due handler che vengono passati come parametri e nel caso non vengona passati aggiungere due handler di default che, anche se non rispondono alla logica applivativa dell'area, evitano che vengono sollevata eccezioni non comprensibili dal Flash Player.

4.9 Una panoramica delle classi utilizzate

4.9.1 Il Modello

Nell'architettura del lato client di *Opendedalo* il *Modello* è la classe actionscript che contiene i dati necessari al *Modulo* in cui viene utilizzata. La classe specifica estende la classe *ModelloBase* che implementa l'interfaccia *IModello*.

L'interfaccia *IModello* contiene tre metodi:

- *set observable(observable:Object): void:* setta l'oggetto che viene gestito dall'*ObserverManager* come osservato
- *get observable(): Object*: analogo al precedente, restituisce l'oggetto che viene usato nell'*ObserverManage*r come osservato
- *get ModelName():String:* restituisce il nome del *Modello*, viene utilizzato nella *Sessione* per ottenere l'insieme di istanze di modelli della stessa classe in uso nell'applicazione.

La classe *ModuloBase* implementa i metodi utilizzati da tutti i modelli e forza la sovrascrittura del metodo *getModelName()*. Dato che in Flex non esiste un costrutto sintattico analogo alla classe astratta di Java viene usato un workaround che consiste nell'implementare il metodo che deve essere sovrascritto con una sola chiamata che solleva un errore, in modo che sia necessario almeno sovrascriverlo nella sottoclasse. Tra i metodi implementati nella classe base vi sono delle chiamate a dei servizi nel back-end che eseguono delle query sul database e restituiscono dei dati relativi alla formattazione dell'applicazione e ai permessi dell'utente loggato. Oltre alle variabili relative a questi metodi di utilizzo generale c'è la viaribile *_observable* di tipo *Object* che rappresenta l'oggetto osservato nell'*ObserverManager*.

La classe *InterrogazioneMovimentiModel* è un esempio di *Modello* specifico. Estende la classe *ModelloBase* e sovracrive il metodo *getModelName()* che restituisce il nome specifico, in questo caso, recuperandolo da una costante definita a livello di classe. Questo è tutto quello che si deve fare per utilizzare l'architettura proposta, gli altri metodi sono specifici della

logica applicativa del modello che si sta sviluppando.

4.9.2 Il Modulo

Il *Modulo* è la classe actionscript che costituisce il cuore dell'architettura lato client. Infatti è la classe che viene caricata da una istanza del *ModuleLoader* quando viene richiesta la pagina dall'applicazione ed è la responsabile dell'inizializzazione e del collegamento delle altre entità coinvolte. Come per il *Modello* il *Modulo* specifico estende una classe base, *ModuloBase*, che implementa un'interfaccia, *IModulo*.

L'interfaccia *IModulo* contiene vari metodi che svolgono attività relative all'internazionalizzazione e per questo vengono implementati nella classe di base ed ereditati da tutti i modulo, l'unico metodo che deve essere implementato dal *Modulo* specifico è *getPresentationModel()*, che restituisce un oggetto che implementa l'interfaccia *IPresentationModel*.

Il costruttore dell classe *ModuloBase* associa all'evento creazione un handler, ovvero una funzione che viene chiamata quando viene completata l'istanziazione. Gli ultimi parametri servono a creare una *weak reference*, che viene gestita in modo più efficiente del garbage collector

```
addEventListener(FlexEvent.CREATION_COMPLETE,creationCompleteHandler,f
alse,0,true);
```

Il metodo *creationCompleteHandler* è standard ed esegue tre chiamate necessarie per ottenere il *PresentationModel* specifico, inizializzarlo e avviarlo.

```
private final function creationCompleteHandler(event :FlexEvent):void{
    presentationModelBase = getPresentationModel();
    presentationModelBase.initPresentationModel();
```

```
        presentationModelBase.startModule();
    }
```

La classe *Modulo* specifica, nell'esempio *ModuloInterrogazioneMovimenti*, è responsabile solo di restituire l'istanza del *PresentationModel* corrispondente.

```
    override public function getPresentationModel():IPresentationModel{
        return new InterrogazioneMovimentiPresentationModel(this);
    }
```

Il parametro this associa il *Modulo* al *PresentationModel*.

4.9.3 Il PresentationModel

Il *PresentationModel* estende una classe base, *PresentationModelBase*, che implementa le interfacce *IPresentationModel* e *Observer*.

L'interfaccia *IPresentationModel* contiene i metodi che recuperano gli oggetti utilizzati dal *PresentationModel* e i metodi per inizializzarlo.

```
    public interface IPresentationModel{
        function startModule():void;
        function initPresentationModel():void;
        function getModello():IModello;
        function getModuleLoader():ModuleLoader;
        function getModuleProxyService():IProxyService
        function getModelName():String;
     }
```

I metodi *startModule()* e *initPresentationModel()* vengono chiamati dal modulo, in particolare dalla classe base che viene estesa dai moduli, quando viene invocato il costruttore della classe modulo particolare.

Il metodo *initPresentationModel()* è uguale per tutti i *PresentationModel* e quindi viene implementato nella classe base.

```
public final function initPresentationModel():void{
    _moduleLoader = getModuleLoader();
    ObserverManager.getInstance().register(_moduleLoader,this);
    _modelloBase = getModello() as ModelloBase;
    _baseProxyService = getModuleProxyService() as ProxyServiceBase;



 _baseProxyService.getModuleServerParameter(
       _moduloBase.getFunctionPrefix());
}
```

Infatti vediamo che non fa altro che ottenere i riferimento del *ModuleLoader*, del *Modello* e del *ProxyService* e, dopo essersi registrato sull'*ObserverManager*, eseguire la chiamata al back-end tramite proxy per ottenere i parametri di base del *Modulo*. Questi parametri sono in realtà sempre gli stessi per tutti i moduli, ma si è preferito utilizzare un metodo parametrico passando un identificatore del modulo caricato per poter gestire sviluppi futuri.

Il metodo *startModule()* è responsabile di inizializzare la pagina mxml associata al modulo e per questo viene sovrascitto nel *PresentationModel* specifico. Nel modulo di esempio esegue solo una chimata al metodo *initFiltroIniziale()* riportato in seguito.

```
public function initFiltroIniziale():void{
    modulo.filtro = new Filtro();
    modulo.addChild(modulo.filtro);
    modulo.filtro.buttonEsci.addEventListener( MouseEvent.CLICK,
       closeModule, false, 0, true);
    modulo.filtro.bottoneTest.addEventListener( MouseEvent.CLICK,
        bottoneTestHandler,false, 0, true);
    modulo.filtro.checkbox.addEventListener( MouseEvent.CLICK,
    abilitaHandler, false, 0, true);
    modulo.filtro.textNome.addEventListener( Event.CHANGE,
      textNomeHandler, false, 0, true);
    modulo.impostaEtichetteInLingua();
}
```

Questo metodo presenta una delle caratteristiche più utili del linguaggio Flex. La classe *Filtro* che viene inizializzata come una normale classe actionscript in realtà è un file mxml contenente la struttura grafica del presentation layer, ovvero dal punto di vista sintattico è una pagina xml costruita usando i tag previsti da Flex. Tuttavia una volta compilata diventa una semplice classe actionscript che può quindi essere gestita in modo trasparente dallo sviluppatore.

L'altro aspetto importante di questo metodo è l'uso delle *weak reference*. Nella vecchia struttura del client Flex gli handler venivano associati in pagina e in questo modo veniva usato il tipo di riferimento forte default del linguaggio. Questo implicava che l'*event dispatcher* evrebbe mantenuto un riferimento tra l'oggetto e la funzione associata e che, quindi, il *garbage collector* non avrebbe mai eliminato nessuno dei due. Usando un riferimento di tipo *weak* invece quando la pagina viene chiusa e l'oggetto eliminato anche

l'*handler* viene cancellato. Dato che l'utilizzo della memoria è uno dei punti critici del flash player questo è un notevole miglioramento, in quanto evita la situazione in cui un utente esauriva la memoria del player dopo aver navigato tra le pagine.

Infine, il metodo *impostaEtichetteInLingua()* gestisce l'internazionalizzazione dalla vista.

4.9.4 L'interfaccia Observer

L'interfaccia *Observer* è costituita dal solo metodo *update*. Questo metodo riceve una stringa che rappresenta l'evento per cui l'oggetto osservatore viene attivato e un *token* che serve a recupere l'istanza corretta dell'osservatore. Nell'applicazione viene usato il *ModuleLoader* come *token*, in quanto è specifico per ogni istanza di modulo che viene avviata.

```
public interface Observer{
   function update(property:String,observable:Object):void; }
```

Quindi nel flusso tipico dell'applicazione l'*ObserverManager* notifica all'osservatore che c'è stata una variazione nel modello, questo in base alla stringa determina quale parte del modello è stata modificata e dopo averlo recuperato dalla *Sessione* aggiorna la pagina nel modo opportuno.

4.9.5 L'ObserverManager

La classe *ObserverManager* è una classe *singleton* che implementa il pattern *Observer*. Praticamente gestiste una lista di coppie osservatore osservato in cui l'oggetto osservato funge da chiave ed è normalmente un'istanza del *ModuleLoader*. L'oggetto osservato è un *PresentationModel*. I metodi che espone sono *register(observable:Object,observer:Observer)* che viene utilizzata dagli osservatori per registrarsi, il metodo per deregistrarsi *unsubscribe(observer:Object, observable:Object)* e il metodo

n*otify(property:String,observable:Object)* che usando il token *observable* recupera l'osservatore corretto e lo attiva chiamando il metodo *notify* e passandogli la stringa *property* che descrive l'evento per cui è stato attivato.

4.9.6 I ServiceProxy

Le classi *ProxyService* fungono da proxy per le chiamate al back-end java tramite gli oggetti remoti esportati da BlazeDs. Le classi specifiche estendono una classe base, *ProxyServiceBase*, che implementa l'interfaccia *IProxyService*.

L'interfaccia *IProxyService* contiene le firme dei metodi comuni a tutte i moduli di *Opendedalo*, quindi metodi che servono a recuperare i parametri della formattazione della pagina, i permessi dell'utente loggato e in generale tutto ciò che è fuori dall'area funzionale del modulo.

Questi metodo sono implementati nella classe *ProxyServiceBase* e vengono invocati dal *PresentationModel* quanto esegue il metodo *initPresentationModel()*.

La classe *ProxyService* specifica del modulo non fa altro che estendere la classe base e contiene un riferimento alla classe *service*, che gestisce il *remote object* specifico del *Modulo*.

```
public function
InterrogazioneMovimentiServiceProxy(modelloBase:ModelloBase{
  super(modelloBase,ServiceFactory.createInterrogazioneMovimentiService());
}
```

Quindi quando viene istanziato il proxy gli viene passato come parametro solo il modello specifico dell'area, mentre la classe service relativa viene iscritta nella classe.

4.9.7 I Service

I *Service* sono le classi che gestiscono gli oggetti remoti esportati tramite BlaseDs, ovvero classi Java i cui metodi possono essere richiamati nel lato client da classi actionscript. Si è

scelto di utilizzare un oggetto remoto per ogni area funzionale del progetto. Come per gli altri elementi le classi service estendono una classe base che implementa un'interfaccia comune.

L'interfaccia *BaseService* contiene i metodi che vengono chiamati dal proxy all'inizializzazione del *PresentationModel*. Questi vengono implementati nella classe base, *FlexBaseService* che, oltre a implementare l'interfaccia *BaseService*, estende la classe *RemoteObject*.

RemoteObject (*mx.rpc.remoting.RemoteObject* per esser più precisi è una classe dinamica, ovvero una classe actionscript a cui possono essere aggiunti metodi e proprietà a runtime) è la classe usata dal linguaggio Flex per dare l'accesso alle classi Java esportate tramite BlazeDs dal codice actionscript. Un oggetto *RemoteObject* viene inizializzato con un riferimento alla destination dell'oggetto esportato. La destination viene utilizzata come *id* per ritrovare l'istanza, ovviamente singleton, della classe lato server che viene esportata.

Nella classe *FlexBaseService* oltre ai metodi di “pubblica utilità” definiti nell'interfaccia è presente il metodo *initOperation(operation:AbstractOperation,handlerFunction:Function=null,handlerFoultFunction:Function=null)* che serve a inizializzare gli *handler* per l'invocazione di un metodo nel *RemoteObject*. I metodi della classe Java nel lato client vengono mappato in oggetti *Operation* (*mx.rpc.AbstractOperation*). Quando un oggetto di questo tipo viene utilizzato, ovvero, quando viene invocato il metodo della classe service Java devono essere gestiti i possibili esiti. Il metodo *initOperation()* serve appunto ad aggiungere due *handler* di default che gestiscano il caso in cui la chiamata vada in errore e quello in cui la chiamata va a buon fine, nel caso non vengano definiti dalla classe *service* specifica del modulo.

5 Conclusioni e considerazioni finali

Il mio giudizio sull'attività di stage è molto positivo. Anche se avevo già una discreta esperienza lavorativa nello sviluppo di web applicazioni non avevo mai lavorato a un progetto così strutturato che rappresenta un buon esempio dello stato dell'arte delle tecnologie utilizzate nel mondo Java Enterprise. Inoltre ho avuto l'opportunità di studiare dei framework molto utilizzati come Spring e Flex interessandomi all'espetto architetturale e non all'implementazione di funzionalità specifiche.

Per quanto riguarda i risultati dello stage la valutazione è stata positiva anche da parte dell'azienda. Infatti l'uso della libreria Spring BlazeDS ha permesso di configurare la comunicazione in modo più semplice senza introdurre regressioni e anzi introducendo delle nuove funzionalità molto utili. La nuova architettura del front-end ha praticamente eliminato i problemi relativi all'uso della memoria e si è dimostrata molto semplice da riutilizzare, pur senza introdurre variazioni sostanziali alla struttura concettuale del presentation layer.

Indice delle figure

6 Bibliografia

[1] Articolo di Rob Johnson sul framework Spring
http://www.theserverside.com/news/1364527/Introduction-to-the-Spring-Framework

[2] Articolo di Martin Fowler sul Dependency Injection.
http://martinfowler.com/articles/injection.html

[3] Articolo di Sony Mathew sull'Inversion of Control
http://www.mokabyte.it/mbtss/2005/06/ioc.htm

[4] Documentazione ufficiale del framework Spring 3.0
http://static.springsource.org/spring/docs/3.0.x/spring-framework-reference/html/

[5] Articolo di S.Rossini e L. Dozio sul pattern MVC
http://www.mokabyte.it/2003/01/pattern_mvc.htm

[6] Documentazione ufficiale del framework Hibernate 4.0
http://docs.jboss.org/hibernate/core/4.0/quickstart/en-US/html/

[7] Documentazione ufficiale del framework Flex 4.0
http://help.adobe.com/en_US/Flex/4.0/UsingSDK/index.html

[8] Documentazione ufficiale del linguaggio ActionScript 3.0
http://help.adobe.com/en_US/FlashPlatform/reference/actionscript/3/index.html

[9] Documentazione ufficiale della libreria Spring BlazeDS Integration 1.5
http://static.springsource.org/spring-flex/docs/1.5.x/reference/html/index.html

[10] Articolo di Angelo Ferraro su BlazeDS
http://www2.mokabyte.it/cms/article.run?articleId=Q1C-IUR-__
USL9DR_7f000001_18359738_eeaf2721

[11] Articolo di Wikipedia sull'IDE Eclipse
http://it.wikipedia.org/wiki/Eclipse_(informatica)

[12] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides
Design Patterns Elements of Reusable Object-Oriented Software
Addison-Wesley (1995)