

ARCHITETTURA TECNICA

Sottosistemi Transazionale e Dati Gestionali (Framework SISN)

Indice

1.	Introduzione	4
1.1	Obiettivi	4
1.2	Struttura del documento.....	4
2	Modello di Riferimento.....	5
2.1	Viste Architetture	5
2.2	Sottosistemi.....	7
3	Architettura esecutiva	10
3.1	Concetti di base.....	10
3.1.1	Architetture multi-tier.....	10
3.1.2	Piattaforma J2EE	11
3.1.3	Il Modello MVC	15
3.2	Framework utilizzati.....	16
3.2.1	Framework Spring MVC	16
3.2.2	Presentation Logic	18
3.2.3	Business Logic	18
3.2.4	Descrizione delle componenti.....	18
3.2.5	Gestire una richiesta con Spring MVC.....	21
3.2.6	Diagramma delle classi - Generale	21
3.3	Architettura del framework NSIS	22
3.3.1	Concetti di base	22
3.3.2	Architettura del framework SISN.....	26
3.3.3	Transazionale - Presentation Logic	27
3.3.4	Transazionale - Business Logic	27
3.3.5	Transazionale - Servizi trasversali.....	32
3.3.6	Transazionale - Files di configurazione	33
3.3.7	Dati Gestionali - DBMS.....	34
3.3.8	Esempio di interazione utente-framework	36
4	Architettura di Sviluppo	38
4.1	Modellazione delle Conversazioni.....	38
4.2	Interazione utente.....	39
4.2.1	Elementi e semantica dell'Interfaccia Utente.....	39
4.2.2	Eventi.....	41
4.3	Gestione del contesto applicativo.....	42

Indice delle figure

Figura 1 - Viste Architettureali	5
Figura 2 - I sottosistemi SISN	8
Figura 3 - Dettaglio dei Sottosistemi	8
Figura 4 - Architettura multi-tier di J2EE	15
Figura 7 - Esempi di Conversazioni	24
Figure 8 - Esempio di struttura di pagina	24
Figura 9 - Esempio di Widgets	25
Figura 10 - Architettura del Framework SISN	26
Figure 11 - Esempio del flusso di una conversazione	30
Figure 12 - Esempio del flusso di una conversazione	31
Figura 13 - Esempio del flusso di una conversazione	36
Figura 17 - Componenti principali della UI	39

1. Introduzione

1.1 Obiettivi

Scopo di questo documento è descrivere le caratteristiche dei sottosistemi Transazionale e Dati Gestionali costituenti il framework architetturale utilizzato per la realizzazione di applicazioni su piattaforma J2EE. Tale framework sarà in seguito denominato **framework SISN**.

Di seguito saranno dapprima richiamati i principi fondamentali su cui si basa il modello architetturale detto e verrà quindi presentata l'**architettura funzionale** del framework SISN, che rappresenta l'insieme delle funzionalità offerte dal framework e utilizzabili nell'ambito dello sviluppo delle componenti software. Successivamente saranno presentate l'**architettura esecutiva** del framework (descrittiva delle sue caratteristiche) e la sua **architettura di sviluppo** (descrittiva dell'insieme degli strumenti di progettazione e sviluppo utilizzabili).

1.2 Struttura del documento

Capitolo		Descrizione
1	Introduzione	Obiettivi e struttura del documento
2	Modello di Riferimento	Presentazione sommaria del modello architetturale di riferimento per lo sviluppo di applicazioni sul SISN
3	Servizi Architetturali	Descrizione dell'architettura funzionale del framework
4	Architettura Esecutiva	Descrizione dell'architettura esecutiva del framework
5	Architettura di Sviluppo	Descrizione dell'ambiente di sviluppo del framework

2 Modello di Riferimento

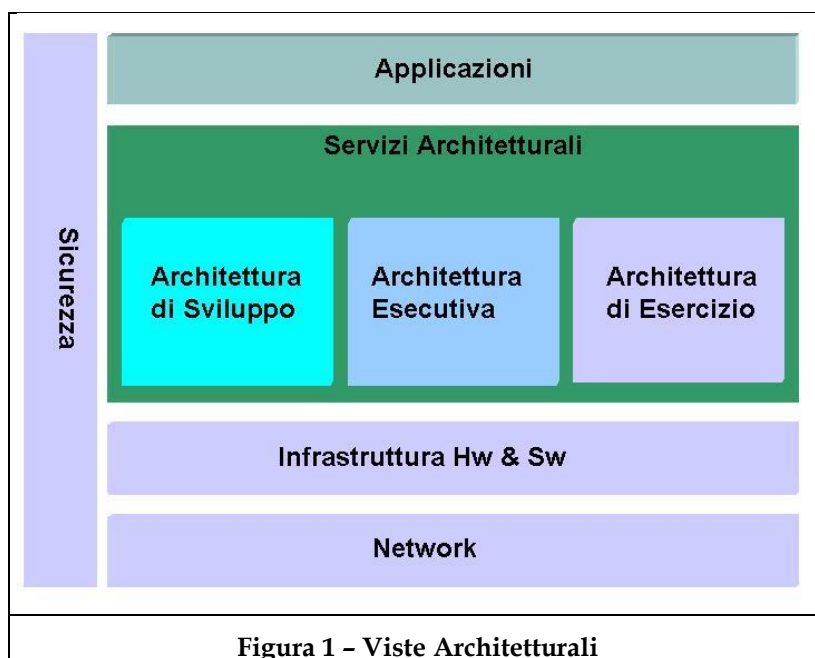
Il **modello architetturale di riferimento** presenta, sulla base degli obiettivi generali caratterizzanti il progetto delle nuove componenti del SISN (obiettivi di governo, di servizio, di comunicazione), le soluzioni architetture, in termini di modelli e tecnologie, più adatte a realizzare un sistema complesso e articolato come il SISN.

Allo scopo di semplificare la complessità insita in un sistema del genere l'architettura nel suo complesso viene descritta e analizzata secondo due ottiche differenti e ortogonali fra loro:

1. suddivisione in **Viste Architetture**
2. suddivisione in **Sottosistemi**

2.1 Viste Architetture

Data la complessità di un sistema come SISN è necessario, al fine di semplificare la descrizione del modello, articolare l'architettura complessiva in un insieme di *viste architetture* secondo quanto riportato nel seguente schema.



- **Applicazioni**: in questa vista sono rappresentate le applicazioni che costituiscono l'interfaccia del sistema SISN verso l'utente finale.
- **Servizi Architetture**: consente di descrivere i servizi che il sistema mette a disposizione dell'utente, fornendo una descrizione degli oggetti che l'architettura rende disponibili per le applicazioni, ma senza entrare nel dettaglio di come saranno implementate.

- **Architettura Esecutiva:** indica le componenti software che costituiscono il framework e le interrelazioni tra esse; è costituita da un insieme di componenti architetturali sui quali viene “costruita” la logica applicativa, indipendentemente dall’infrastruttura tecnologica adottata.
- **Architettura di Sviluppo:** questa vista è costituita da un insieme di strumenti/servizi che consentono di gestire lo sviluppo di nuove applicazioni per il SISN. Tali strumenti comprendono i sistemi di sviluppo, i sistemi di gestione della configurazione del sistema, i tools metodologici di sviluppo delle applicazioni (es. Rational Rose, ecc...).
- **Architettura di Esercizio:** è costituita da un insieme di strumenti/funzioni a supporto dell’operatività quotidiana, in grado di garantire un maggior livello di qualità operativa delle applicazioni nell’ambiente di esercizio (es. tracciatura delle anomalie di produzione, verifica dei carichi delle macchine, bilanciamento, eventuale integrazione con tools di Service&Network Mgmt aziendali).
- **Infrastruttura:** vista che descrive la piattaforma hardware e software, in termini di prodotti e sistemi su cui le componenti individuate nelle precedenti viste dell’architettura (in particolare esecutiva, di sviluppo, di esercizio) sono implementate.
- **Network:** rappresenta l'insieme di computer che compongono la rete e che possono anche trovarsi in siti remoti.
- **Sicurezza:** descrive gli aspetti relativi alla sicurezza informatica delle applicazioni sviluppate per il SISN. In tale ambito rientrano le problematiche di autenticazione degli utenti, profilazione delle applicazioni, sicurezza delle informazioni scambiate (crittografia dei dati), uso di tecnologie proprie delle tematiche di sicurezza (ad esempio certificati e firma digitale).

2.2 Sottosistemi

La suddivisione in sottosistemi è una vista dell'architettura del tutto ortogonale alla precedente e dettata da un approccio più "funzionale", ovvero dall'analisi delle tipologie di applicazioni che fanno parte del SISN.

Restano definiti sei sottosistemi, ognuno dei quali è specializzato nell'erogazione di una particolare categoria di servizi (cfr. figura 2), come di seguito indicato.

- **Sottosistema Transazionale:** è il sottosistema che fornisce servizi (servizi di Front End, gestione dell'Accesso, ecc.) necessari ad applicazioni che richiedono una stretta interazione tra gli utenti e il sistema SISN, quali le elaborazioni sui dati, garantendo rapidi tempi di risposta.
- **Sottosistema Dati Gestionali:** è preposto alla gestione dei dati del SISN relativi alle componenti gestionali. Tale sottosistema è utilizzato dalle applicazioni transazionali, dalle componenti di gestione dei contenuti e dalle applicazioni batch. E' la fonte primaria di alimentazione del sottosistema di Data Warehousing, ma è logicamente disaccoppiato da esso.
- **Sottosistema di Datawarehousing:** è il sottosistema al quale è demandato il compito di gestire (caricare ed analizzare) ingenti moli di dati provenienti da i sistemi interni al Servizio Sanitario Nazionale e da sistemi esterni di interesse per l'Amministrazione.
- **Sottosistema di Cooperazione:** permette e controlla l'esecuzione di processi complessi che coinvolgono sistemi informativi di differenti domini (Ministero, Regioni, realtà territoriali, etc).
- **Sottosistema di Sicurezza:** comprende i servizi di gestione della sicurezza comuni ai diversi sottosistemi, ad esempio l'accesso da parte degli utenti al sistema SISN si realizza attraverso la centralizzazione della gestione degli accessi e del profiling applicativo.

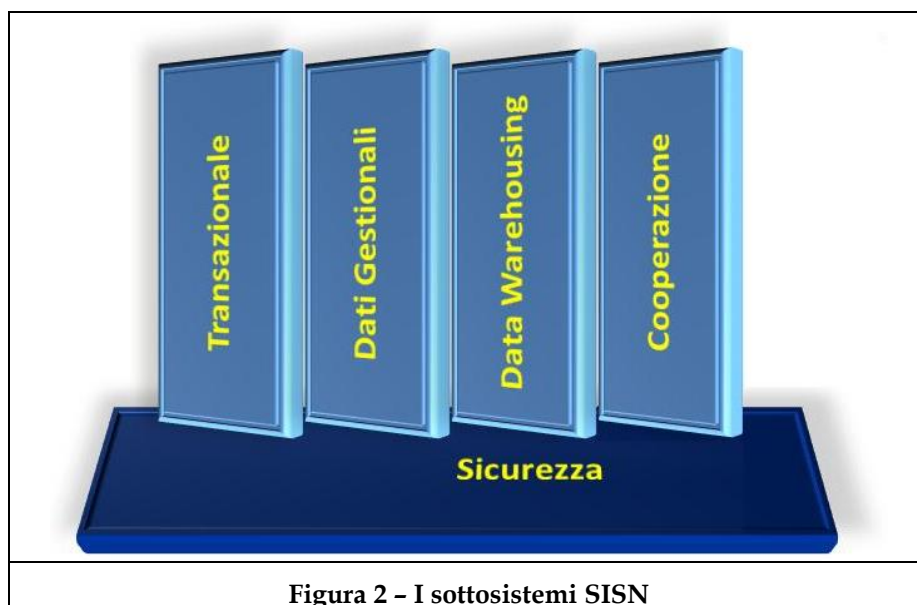


Figura 2 – I sottosistemi SISN

Ogni sottosistema inoltre, nell'ambito della sua specificità, può essere visto come composto da un insieme di componenti funzionali specializzati; la figura seguente, per l'appunto, mostra l'insieme dei sottosistemi con un dettaglio di tali componenti.

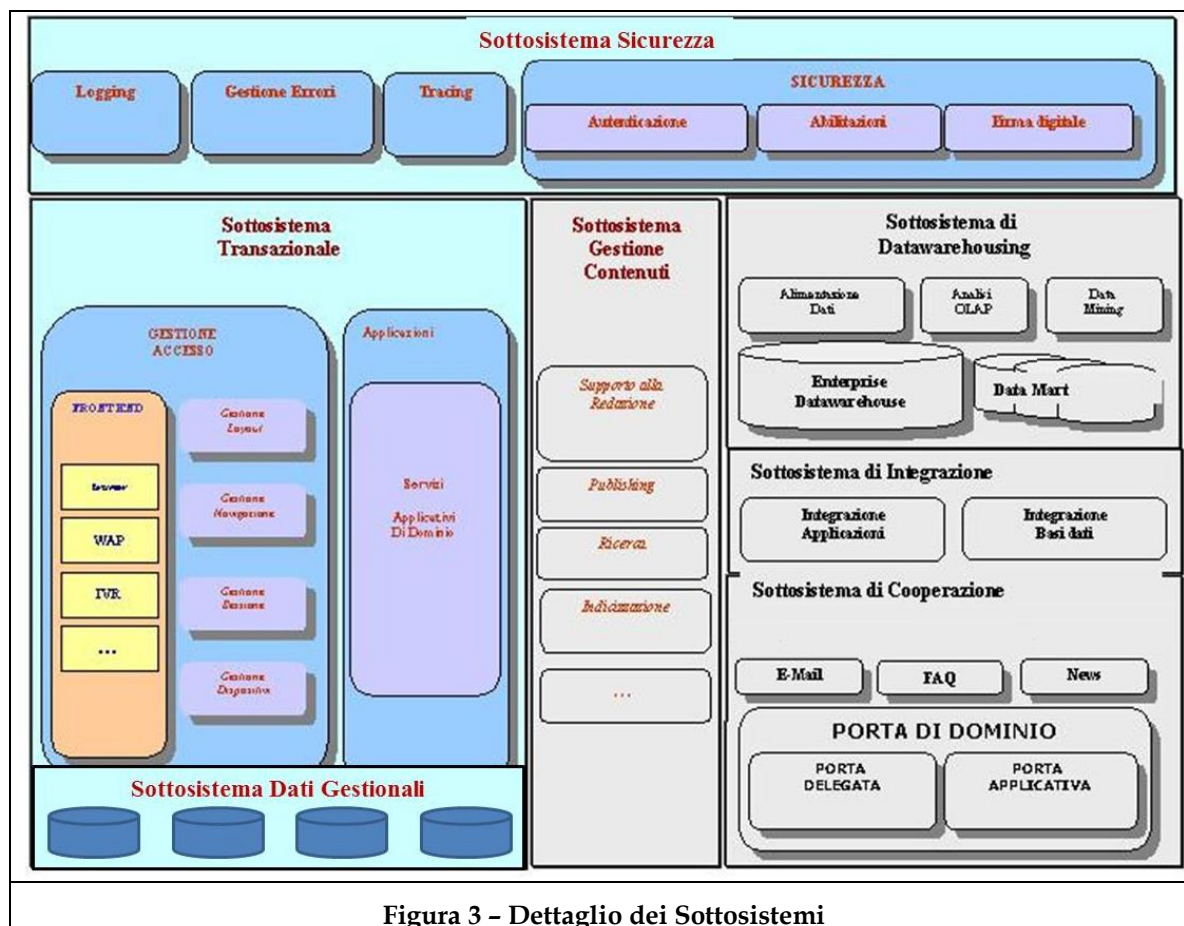


Figura 3 – Dettaglio dei Sottosistemi

La precedente figura evidenzia l'esistenza di componenti "infrastrutturali", sia trasversali che specifici del singolo sottosistema, che risultano non legati alla singola applicazione e che, di conseguenza, possono essere implementati in un **framework**. Un framework è un'applicazione "semi-completa", riutilizzabile, e che può essere specializzata per produrre applicazioni specifiche. Fra i principali motivi che giustificano lo sviluppo e l'adozione di un framework spiccano la riduzione dei costi ed e l'aumento della qualità del software.

Il framework SISN copre completamente le funzionalità previste dal Sottosistema Transazionale e dai Servizi Trasversali e Dati Gestionali.

Lo sviluppo delle applicazioni attraverso l'impiego del framework comporta i vantaggi di seguito evidenziati.

- **Funzionalità** – Un framework architetturale è lo scheletro di una applicazione, scritto con lo scopo di essere usato come base per il rapido sviluppo di una famiglia di applicazioni. Esso cattura le parti comuni della famiglia, consentendo di aggiungere specifiche funzionalità in modo semplice e veloce. L'architettura del framework è composta da tutti i servizi automatizzabili che implementano i requisiti funzionali, e che permettano l'integrazione con applicazioni esterne. Essa descrive la forma di un'applicazione e come questa implementa i requisiti funzionali.
- **Affidabilità** – Grazie all'isolamento delle funzionalità infrastrutturali offerte dal framework, dalla parte funzionale delle applicazioni, si riduce notevolmente il rischio di possibili anomalie nei sistemi realizzati. L'infrastruttura offerta dal framework è infatti una componente ampiamente collaudata e riutilizzabile su differenti applicazioni.
- **Modularità** – I dettagli implementativi vengono nascosti da interfacce stabili. La localizzazione degli impatti dovuti ai cambiamenti della progettazione o dell'implementazione migliora la qualità del software prodotto e riduce lo sforzo richiesto per comprendere e mantenere quello esistente.
- **Riusabilità** – La stabilità delle interfacce fornite da un framework consente la definizione di componenti generici che possono essere riutilizzati per creare nuove applicazioni. Il riuso di componenti di un framework può portare ad un sostanziale miglioramento della produttività dell'attività di programmazione, oltre ad aumentare qualità, prestazioni, affidabilità ed interoperabilità del software.

3 Architettura esecutiva

In questo capitolo si descrive l'architettura esecutiva del framework, ovvero la descrizione dei componenti tecnologici che lo compongono.

Innanzitutto è opportuno richiamare alcuni concetti generali (modelli e tecnologie) che stanno alla base della costruzione del framework:

1. Architetture multi-tier
2. Piattaforma J2EE
3. Modello MVC

Successivamente verrà presentata l'architettura del framework in termini dei suoi principi ispiratori e dei principali componenti costituenti.

3.1 Concetti di base

3.1.1 Architetture multi-tier

Una tipica classificazione delle architetture è quella che considera il numero di livelli su cui viene realizzata un'applicazione, dove, per **livello**, si intende un gruppo di componenti, software e/o hardware, omogeneo per tipologia di servizi che implementa. La suddivisione in livelli è fatta principalmente per garantire *scalabilità* e *sicurezza* ma anche per facilitare lo sviluppo e la manutenzione delle applicazioni.

Le principali architetture sono a due livelli (2-tier), a tre livelli (3-tier) o a N livelli (N-tier), con N che rappresenta il numero di livelli.

3.1.1.1 Architettura a due livelli

L'**architettura a due livelli** è costituita dal tradizionale *modello client/server* molto diffuso a partire dai primi anni '90 e ancora oggi utilizzato. Un'applicazione viene suddivisa in due componenti logicamente separati, ciascuno dei quali svolge mansioni specifiche:

- un *Client* o *front-end*, si occupa di gestire l'interazione con l'utente dell'applicazione
- un *Server* o *back-end*, si occupa di gestire i dati trattati.

Tipicamente esistono un server centralizzato e differenti client connessi al server per mezzo di una rete. Il client comunica con il server per mezzo di un protocollo di rete.

Nell'ambito di questo modello esistono in genere due interpretazioni:

- la prima vede il server coincidere con un DBMS, pertanto il client è responsabile oltre che dell'interazione utente anche dell'invocazione dei servizi di accesso ai dati (cioè dal client sono ad esempio eseguite le istruzioni SQL);
- nella seconda il server è implementato come un processo che resta in attesa di una richiesta di servizio (ad esempio utilizzando il protocollo RPC) ed è lui che si occupa di implementare le operazioni di accesso al DBMS.

3.1.1.2 Architettura a tre livelli

I limiti dell'architettura a due livelli sono di due tipi:

- ogni volta che viene distribuita una nuova versione dell'applicazione, occorre aggiornare tutti i client.
- il client contiene tutte le componenti applicative oltre i dati, per questo motivo si parla, in senso dispregiativo, di *fat client*; di conseguenza il codice del client risulta complesso da sviluppare e da mantenere.

Per superare questi limiti il modello client/server si è evoluto nel modello di **architettura a tre livelli** che prevede l'aggiunta di un nuovo *livello intermedio*, tra il client ed il server; conseguentemente parte dei componenti che prima era sviluppati nel client trova posto in questo livello intermedio. I livelli pertanto diventano:

- **Client o Presentation Logic**: è la componente applicativa che si occupa della gestione dell'interazione utente;
- **Application Server o Application Logic**: contiene la logica applicativa e lo stato;
- **Data Server o Data Access Logic**: implementa la logica di accesso ai dati.

In questo modo il problema della distribuzione del software relativo ai client si riduce notevolmente, in quanto il software diventa molto più semplice e di conseguenza più facile da installare e configurare.

Dal punto di vista dello sviluppo e della manutenzione del software, invece, la creazione del livello intermedio porta ad una maggiore modularizzazione e ad una più efficiente organizzazione del gruppo di lavoro.

3.1.1.3 Architettura N-livelli

Le architetture a n-livelli con un numero di livelli maggiore di 3 sono variazioni dell'architettura a tre livelli derivanti soprattutto dal modo con cui si suddivide il livello intermedio. L'esigenza di suddivisione in ulteriori livelli è legata principalmente a due esigenze:

- **Efficienza nello sviluppo e manutenzione**: suddividendo un sistema in più strati, ognuno dei quali ha una singola responsabilità, ogni risorsa del gruppo di lavoro sarà assegnata a seconda delle proprie competenze allo sviluppo di componenti che appartengono ad un solo livello;
- **Scalabilità**: la distribuzione di un'applicazione su differenti strati consente un miglior sfruttamento delle tecniche di elaborazione a *cluster*, con un conseguente miglioramento della scalabilità.

3.1.2 Piattaforma J2EE

La piattaforma **J2EE (Java 2 Enterprise Edition)** fornisce una serie di tecnologie, basate su Java e su un approccio *component-based*, che consentono la progettazione, lo sviluppo, l'assemblaggio e la

distribuzione di applicazioni. La piattaforma utilizza un modello di architettura multi-tier: un'applicazione è divisa in componenti a seconda delle funzioni svolte e i vari componenti sono installati su differenti macchine a seconda dello strato a cui appartengono.

3.1.2.1 Contenitori

Un concetto che assume un ruolo centrale nell'ambito della piattaforma J2EE è quello di **containers** (contenitori). I contenitori sono *ambienti runtime standardizzati* che hanno lo scopo di offrire determinati servizi ai componenti della piattaforma J2EE in modo da garantire la gestione del ciclo di vita del componente; ogni specifico contenitore offre dei servizi specializzati tipici del tipo di componente che gestisce. I componenti, quindi, possono fare affidamento sull'esistenza di tali servizi in quanto devono essere resi disponibili da qualsiasi prodotto di mercato che segua le specifiche della piattaforma J2EE.

Attraverso l'uso di **deployment descriptors** (file XML che specificano il comportamento a tempo di esecuzione del componente e del contenitore che lo ingloba), i componenti possono essere configurati per l'ambiente di uno specifico contenitore piuttosto che implementare tali caratteristiche direttamente nel codice. Le proprietà che possono essere configurate a tempo di distribuzione (*deployment time*) includono controlli sulla sicurezza, controllo transazionale e altro.

3.1.2.2 Applicazioni

Concettualmente una **J2EE Application** coincide con quello che gli utenti chiamano applicazione o sistema; dal punto di vista tecnologico essa è composta da moduli (**J2EE Module**) e/o componenti (**J2EE Component**).

Una J2EE Application, a deployment time, è costituita da un unico file JAR con estensione **.EAR** (**Enterprise Archive**).

3.1.2.3 Componenti

Un **J2EE Component** è un'unità software supportata da un apposito contenitore. La piattaforma J2EE definisce quattro tipi di contenitori:

- **EJB (ENTERPRISE JAVA BEANS) CONTAINER**
- **WEB COMPONENT CONTAINER**
- **APPLICATION CLIENT CONTAINER**

□ Un **EJB (ENTERPRISE JAVA BEAN)**, è un componente utilizzato per lo sviluppo della logica applicativa di un'applicazione; è possibile pensarlo come un *building block* che può essere usato da solo oppure assieme ad altri EJB su un server J2EE. Esistono tre tipi di EJB:

- **Session Beans**
- **Entity Beans**
- **Message-driven Beans**

Un EJB Container è fornito da un *server EJB* oppure da un *server J2EE* (ad esempio, IBM WebSphere, BEA WebLogic, JBOSS, ...etc).

□ Un **WEB COMPONENT**, è un componente che fornisce servizi che rispondono alle richieste provenienti dal client. Esistono due tipi di componenti di questo tipo:

- **Servlets**: sono classi Java che processano dinamicamente richieste (tipicamente http) provenienti dal client e costruiscono una risposta (ad esempio una pagina HTML);
- **JSP (Java Server Pages)**: sono documenti testuali eseguiti come servlet ma che offrono un approccio più naturale, rispetto alle servlet, per la costruzione di contenuto in parte statico.

Sebbene le servlet e le pagine JSP possono essere intercambiabili, ognuna di essa ha un proprio ruolo. Le servlet sono da preferire quando si deve gestire il controllo funzionale di un'applicazione, come il *dispatching* di richieste o la gestione di dati non testuali. Le pagine JSP risultano più appropriate per la generazione di contenuto testuali "a tag" quale HTML, SVG, WML e XML.

□ Un **Application Client**, è un componente scritto in Java che viene eseguito su un client in una JVM (Java Virtual Machine). Le applicazioni client hanno accesso ad un insieme delle API messe a disposizione dalla piattaforma J2EE (JNDI, JDBC, RMI-IIOP, JMS).

3.1.2.4 Moduli

Un **J2EE Module** (modulo) è un'unità software consistente in uno o più *J2EE Component* dello stesso tipo di *container* e di un *deployment descriptor* di quel tipo. I moduli possono essere distribuiti singolarmente (modalità *stand alone*) oppure assemblati in una *J2EE Application*. Esistono tre tipi di moduli:

- **EJB Module**: sviluppato come un file EJB JAR, con estensione *.jar*, è l'unità minima sviluppabile ed utilizzabile di un *enterprise bean*.
- **Web Module**: sviluppato come un archivio Web (file WAR), si tratta di un file JAR con estensione *.war*. È l'unità minima sviluppabile ed utilizzabile di risorse web.
- **Application Client Module**: sono « impacchettati » come file JAR con estensione *.jar*. Contiene classi java che implementano il client e un *deployment descriptor* di un'applicazione client.

3.1.2.5 Modello multi-tier

La piattaforma J2EE è stata progettata per dare supporto, sia sul lato client che sul lato server, allo sviluppo di applicazioni distribuite organizzate secondo un **modello architetturale a più livelli**.

I tipici livelli previsti dalla piattaforma J2EE sono i seguenti:

- un **Client Tier**, che fornisce l'interfaccia utente;
- uno o più **Middle Tier**, che fornisce servizi ai client e implementa la logica applicativa;
- un **EIS (Enterprise Information Systems) Tier**, che implementa la logica di accesso ai dati (DBMS o sistemi legacy).

La figura seguente illustra i vari componenti e servizi che costituiscono un tipico ambiente J2EE.

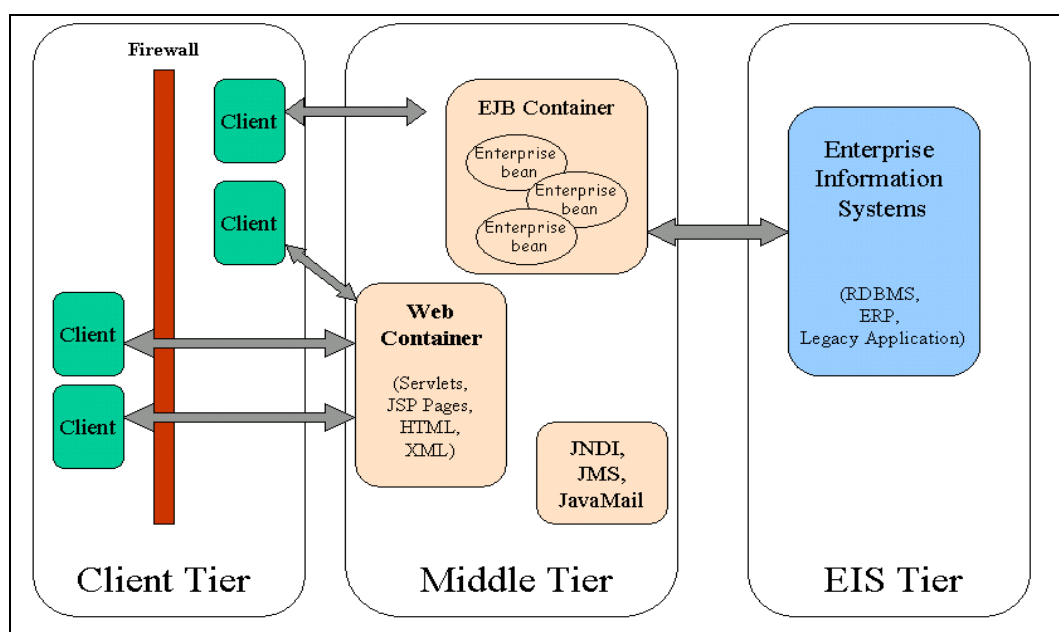


Figura 4 – Architettura multi-tier di J2EE

La figura evidenzia che il *client tier* supporta vari tipi di client, sia internamente che esternamente ad un firewall aziendale. Il *middle tier* fornisce i servizi al client per mezzo del *Web Container* risiedente nel sottolivello **Web Tier**, e supporta i componenti che implementano la logica applicativa grazie all'*EJB Container* presente nell'**EJB Tier**. Il back-end costituito dall'EIS Tier è accessibile attraverso API standard (ad esempio JDBC).

3.1.3 Il Modello MVC

L'intento del pattern architetturale **Model View Control (MVC)** è di separare il più possibile tra loro le parti dell'applicazione adibite al controllo, all'accesso ai dati e alla presentazione.

Questo approccio porta ad innegabili vantaggi come:

- indipendenza tra i business data (model) la logica di presentazione (view) e quella di controllo (controller) ;
- separazione dei ruoli e delle relative interfacce ;
- viste diverse per il medesimo model;
- semplice supporto per nuove tipologie di client: bisogna scrivere la vista ed il controller appropriati riutilizzando il model esistente.

Con il termine **Model** si individua la rappresentazione dei dati dell'applicazione enterprise e le regole di business con cui tali dati vengono acceduti e modificati.

La **View** è la vista del modello. Uno stesso modello può quindi essere presentato secondo diverse viste.

Il **Controller** è l'entità che interpreta le richieste della view in azioni che vanno ad interagire con il Model aggiornando conseguentemente la view stessa.

L'architettura MVC separa gli obiettivi della progettazione, riduce la duplicazione del codice, centralizza il controllo, e rende l'applicazione più facile da modificare. Ciò consente a sviluppatori con *skills* diversi di concentrarsi sulle loro specifiche competenze e collaborare attraverso interfacce chiare e definite. Il controllo su alcune caratteristiche dell'applicazione, come ad es. la sicurezza, il *logging* e il flusso fra le viste, può essere centralizzato.

L'implementazione tipica del *pattern* MVC con tecnologia J2EE nella costruzione di un *application framework web* viene comunemente indicata con il nome *Modello 2*, in cui una *servlet*, con responsabilità di *Front Controller*, gestisce la comunicazione con il *client* e l'esecuzione della logica di *business*, mentre la presentazione è realizzata principalmente attraverso pagine JSP. Il termine *Modello 1* viene invece usato per indicare un'architettura in cui il *client* invoca direttamente le pagine JSP senza la mediazione di un *controller*.

Nel *Modello 2* la *servlet* di controllo centralizza la logica per l'instradamento delle richieste basata sull'URL della richiesta, i parametri di input e lo stato dell'applicazione.

Per l'implementazione del modulo *View* vengono tipicamente usate pagine JSP per produrre contenuti di tipo testuale come HTML o XML, mentre sono preferibili le *serolet* per generare contenuti binari (RTF, PDF) o a struttura variabile.

Il modulo *Model* che rappresenta sia i dati che la logica di *business* può essere implementato con *Enterprise JavaBean* per soddisfare requisiti come scalabilità, concorrenza, bilanciamento del carico, gestione automatica delle risorse, accesso a logica e dati di *business* condivisi, oppure con *JavaBean* standard per un accesso ai dati semplice e meno critico.

La separazione della logica di *business* dalla presentazione comporta i vantaggi di seguito indicati.

- **Minimizzazione degli impatti dei cambiamenti** – le regole di *business* possono cambiare con modifiche minime od assenti al modulo di presentazione. Viceversa, la presentazione od il *workflow* dell'applicazione può cambiare senza influenzare il modulo di *business*.

- **Maggiore manutenibilità** – la modifica, ad esempio, dei componenti di *business*, delle direttive di *include* lato *server*, *custom tag*, *stylesheet*, altera il comportamento dell'applicativo in qualunque punto essi vengano referenziati.

- **Indipendenza dal client e riuso del codice** – la logica di *business* disponibile come semplici chiamate a metodi di oggetti di *business* può essere usata da tipi di *client* diversi.

- **Separazione dei ruoli degli sviluppatori** – la separazione della logica di *business* e la presentazione consente agli sviluppatori di concentrarsi sulla loro area di competenza: presentazione dei dati, elaborazione della richiesta, regole di *business*.

3.2 Framework utilizzati.

La maggiorparte dei sistemi nell'ambito NSIS sono state realizzate utilizzando un framework specifico per il Ministero (**framework NSIS descritto nel paragrafo 3.3**), ma a partire dal 2015 per la realizzazione di nuove applicazioni è stato adottati dei framework e dei tool open di uso comune come **Spring MVC, Bootstrap, Hibernate, Ibatis**.

I due aspetti fondamentali per cui si è scelto di utilizzare Spring sono la sicurezza e l'accesso ai dati.

- Sicurezza.
Spring ha al suo interno un modulo dedicato alla sicurezza chiamato Spring Security che consente la gestione dell'autenticazione e delle autorizzazioni degli utenti.
- Accesso ai dati.
Sfruttando appieno le potenzialità offerte dalla Dependency Injection e dall'Aspect Oriented Spring consente di integrarsi con framework che si occupano della persistenza dei dati come Hibernate che partendo dalla struttura del data base consente di generare l'insieme di java bean necessari per la gestione della corretta persistenza dei dati.

3.2.1 Framework Spring MVC

L'architettura del framework che verrà utilizzato per l'implementazione del sistema in oggetto, si presenta come una architettura 3-tier. Dal punto di vista tecnologico, essa utilizza la piattaforma J2EE e si basa sul pattern MVC (Model View Control).

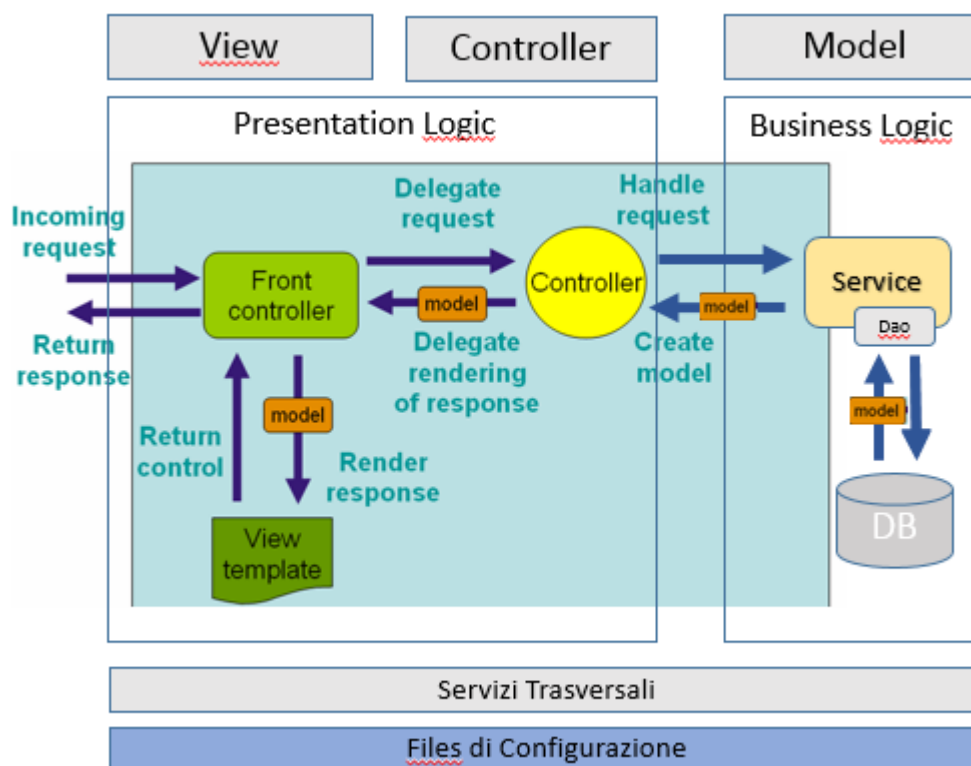


Figura 5 – Architettura Spring MVC

Come si può vedere dalla figura, il framework Spring MVC è costituito dai seguenti livelli:

- 1 **Presentation Logic** - è il livello che gestisce l'interazione con l'utente.
- 2 **Business Logic** - è il livello che implementa le funzionalità.
- 3 **Servizi trasversali** - si tratta di una serie di servizi in comune che possono essere utilizzati da tutti i componenti.
- 4 **Files di configurazione** - è l'insieme dei files la cui configurazione permette di usufruire di altre importanti caratteristiche del framework.

Vengono di seguito elencate le corrispondenze tra le componenti applicative e i livelli MVC

Modello MVC	Associazione Con
VIEW	Pagine JSP
CONTROLLER	DispatcherServlet e Controller
MODEL	Service, Dao e Java Bean

3.2.2 Presentation Logic

Il livello di *Presentation Logic* costituisce la parte front-end dell'applicazione. Essa comprende il livello View e Control del modello MVC e provvede quindi sia a "presentare" l'applicazione all'utente, sia a gestirne le funzionalità. Il livello è costituito dalle seguenti componenti:

- **JSP** - il componente è chiamato a gestire l'interfaccia utente, attraverso l'utilizzo di pagine HTML generate dinamicamente tramite *JSP*. Tutte le richieste effettuate tramite JSP e visualizzate tramite un browser web, passano attraverso il componente *DispatcherServlet*.
- **DispatcherServlet** - esso viene implementato attraverso una servlet HTTP J2EE ed ha il compito di identificare il dispatcher al fine di indirizzare il flusso verso il controller demandato ad eseguire le funzionalità richieste. Il componente è parte costitutiva del framework e può essere configurato attraverso opportuni file XML di configurazione.
- **Controller** - nel mezzo tra Model e View, amministra le richieste in arrivo e reindirizza le risposte al View

3.2.3 Business Logic

Il livello esegue le funzionalità di business. Esso viene implementato attraverso il componente *Action*. Esso è l'oggetto demandato ad evadere per intero la richiesta di esecuzione di una funzionalità.

Il livello è costituito dalle seguenti componenti:

- **Action** - E' la classe di business che evade per intero la richiesta effettuata dall'utente tramite una pagina JSP. Ogni funzionalità descritta nei casi d'uso (per cui si rimanda al documento di "Requisiti Funzionali") necessita l'implementazione di una *Action*.
- **DAO (Data Access Object)** - Gli oggetti istanziati sono invocati dal componente *Service*. Essi hanno il compito di effettuare l'accesso ai dati per effettuare operazioni di lettura e di inserimento dati. Costituisce il livello di comunicazione tra la banca dati e il sistema applicativo.
- **Model** - E' l'oggetto (Java Bean) contenente i dati di business richiesti dall'utente tramite le azioni invocate nel livello di *Presentation Logic*.

3.2.4 Descrizione delle componenti

3.2.4.1 DispatcherServlet

Come tutte le servlet, anche la *DispatcherServlet* va configurata nel file *web.xml*, il file descriptor della web application. Nel caso di Apparecchiature Sanitarie la dispatcher è rappresentata dalla *ControlServlet*, esso appare così:

```
<web-app>
<display-name>GrandiApparecchiature</display-name>
  <servlet>
    <servlet-name>ConfigServlet</servlet-name>
    <servlet-class>com.engiweb.framework.configuration.ConfigServlet</servlet-class>
    <init-param>
      <param-name>AF_CONFIG_FILE</param-name>
      <param-value>/WEB-INF/conf/master.xml</param-value>
    </init-param>
  </servlet>
</web-app>
```

```
        </init-param>
        <load-on-startup>0</load-on-startup>
    </servlet>
    <servlet>
        <display-name>ControlServlet</display-name>
        <servlet-name>ControlServlet</servlet-name>
        <servlet-
            class>com.accenture.grandiapparecchiature.dispatcher.ControlServlet</servlet-
class>
    </servlet>

</web-app>
```

Da notare le definizioni del parametro `contextConfigLocation` e del listener `ContextLoaderListener`, necessari rispettivamente per localizzare il file *applicationContext.xml* contenente le dipendenze del progetto e caricare all'avvio l' `Application Context`.

La servlet `DispatcherServlet` così configurata, denominata `ControlServlet`, verrà richiamata per tutte le richieste alle pagine il cui nome corrisponde all'espressione `*.htm`.

Durante la sua inizializzazione, essa carica un particolare file xml, all'interno del quale sono configurati i vari controller, handler mapping e resolver coinvolti nell'elaborazione delle richieste ad essa associate. Il file *applicationContext.xml* per L'IoC di Spring, definisce ciò che viene chiamato `WebApplicationContext` associato alla servlet `ControlServlet`.

I controller permettono di accedere al comportamento della web application. In Spring MVC una classe che voglia agire da controller deve implementare l'interfaccia `Controller` o essere marcata con l'annotazione `@Controller`:

```
public interface Controller {
    ModelAndView handleRequest(
        HttpServletRequest request,
        HttpServletResponse response) throws Exception;
}
```

Essendo molto generica, il framework mette a disposizione dello sviluppatore una serie di Controller specificatamente pensati per risolvere alcune situazioni comuni. In Apparecchiature Sanitarie ad esempio tutte i controller estendono la classe *BaseMultiActionFormController*, la quale estende a sua volta il controller fornito da Spring *MultiActionFormController*. Esso permette ad un unico controller di gestire più richieste, siano esse di tipo GET o POST, centralizzando funzionalità comuni su un'unica classe.

I controllers, così come le servlet, sono mappati ad una o più URL e hanno accesso agli oggetti

HttpServletRequest e *HttpServletResponse* necessari per poter estrarre i dati inviati dall'utente, elaborarli (attraverso i servizi messi a disposizione dallo strato di business-logic sottostante) ed infine indirizzare i risultati alla view dedicata alla loro rappresentazione.

3.2.4.2 URL MAPPING

Per determinare quale controller deve elaborare una particolare richiesta, effettuando quindi ciò che viene definito come *URL Mapping*, bisogna configurare un *HandlerResolver*. Spring, a

seconda della strategia utilizzata per il mapping, mette a disposizione alcune implementazioni di *HandlerResolver* : nel caso di Apparecchiature Sanitarie si è scelto di usare il *InternalPathMethodNameResolver*, attraverso il quale Spring mappa automaticamente gli URL ai metodi dei Controller tramite il nome del Controller stesso. Tali metodi devono restituire un oggetto *ModelAndView* affinché il framework sia poi in grado di determinare quale pagina restituire all'utente.

3.2.4.3 VIEW RESOLVER

L'oggetto *ModelAndView* contiene al suo interno un riferimento alla view da visualizzare e i risultati dell'operazione richiesta. Tale riferimento non è altro che un nome logico, che verrà poi

risolto da un oggetto, il *ViewResolver* appunto, fornito direttamente da Spring MVC.

Questa separazione permette un completo disaccoppiamento tra il Controller e la particolare tecnologia di *rendering* utilizzata per la view.

La tecnologia di rendering predefinita per Spring MVC è JSP, ma nulla vieta di utilizzare altre tecnologie, come XML, JSON, JSF. Per i sistemi NSIS abbiamo deciso di utilizzare JSON.

3.2.4.4 HIBERNATE

Hibernate è uno strumento di object/relational mapping open source per Java che consente di sviluppare classi persistenti e logica di persistenza senza preoccuparsi di come gestire i dati. Hibernate non solo si prende cura della mappatura da classi Java con le tabelle del database e da tipi di dati Java a tipi di dati SQL, ma fornisce anche le query e le strutture di recupero dei dati. Lo sviluppatore viene pertanto esonerato dall'implementazione delle classi DAO per il recupero manuale dei dati con il semplice utilizzo di SQL e JDBC . Di seguito schema generale sul funzionamento di Hibernate.

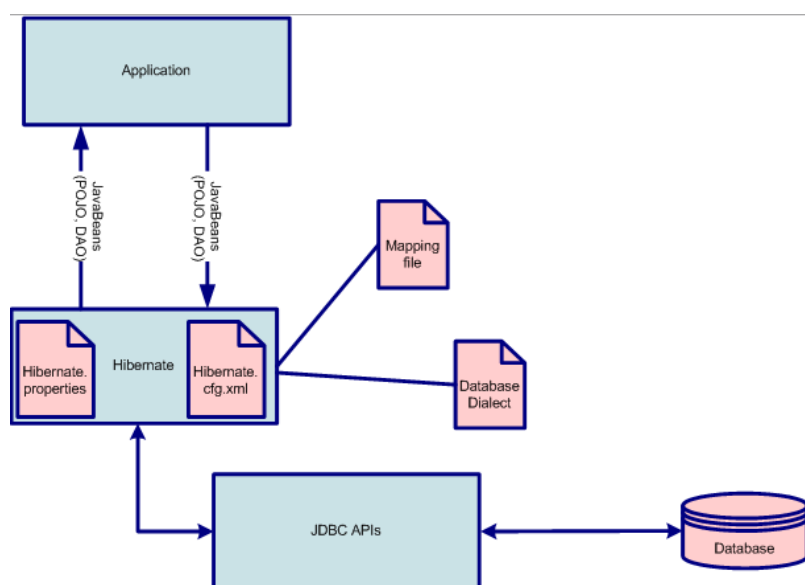


Figura 6 - Hibernate

3.2.5 Gestire una richiesta con Spring MVC

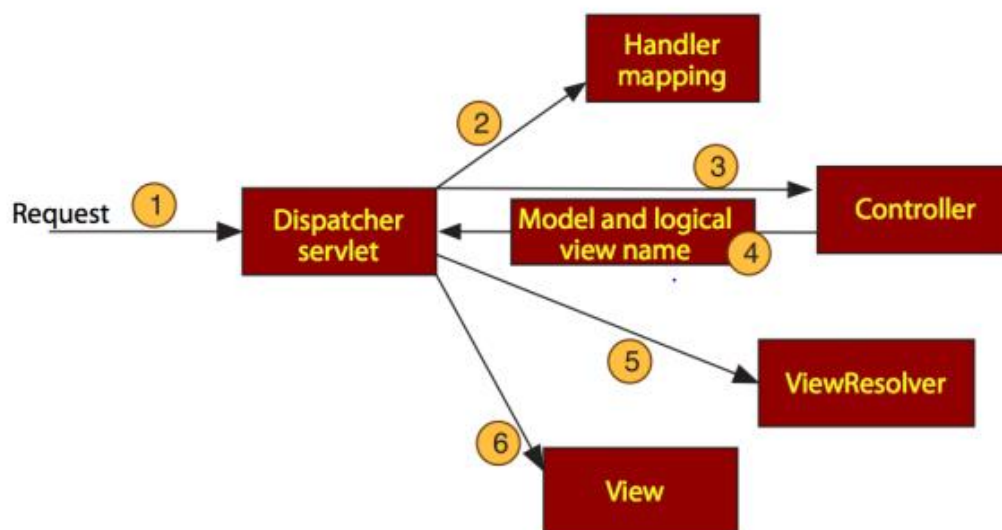


Figura 7 -Gestione delle richieste in Spring

Ogni volta che un utente seleziona un link o dà il comando di invio in un determinato form, effettua una richiesta alla web application. Con Spring MVC, tale richiesta viene letta inizialmente dal DispatcherServlet che delega la responsabilità della sua gestione al Controller appropriato. Solitamente in una web application esso non è unico, così il DispatcherServlet si affida all'Handler Mapping per riconoscere il destinatario, secondo una mappa che associa l'URL della richiesta ai controller. Una volta identificato il DispatcherServlet gli invia la richiesta. Arrivata a destinazione, la richiesta viene eseguita integrandosi con il model, e saranno recuperate le informazioni da visualizzare all'utente. Il Controller restituirà tali informazioni, sotto forma di ModelAndView al DispatcherServlet che la inoltra prima al View Resolver, il quale sceglierà la pagina corretta da visualizzare.

3.2.6 Diagramma delle classi - Generale

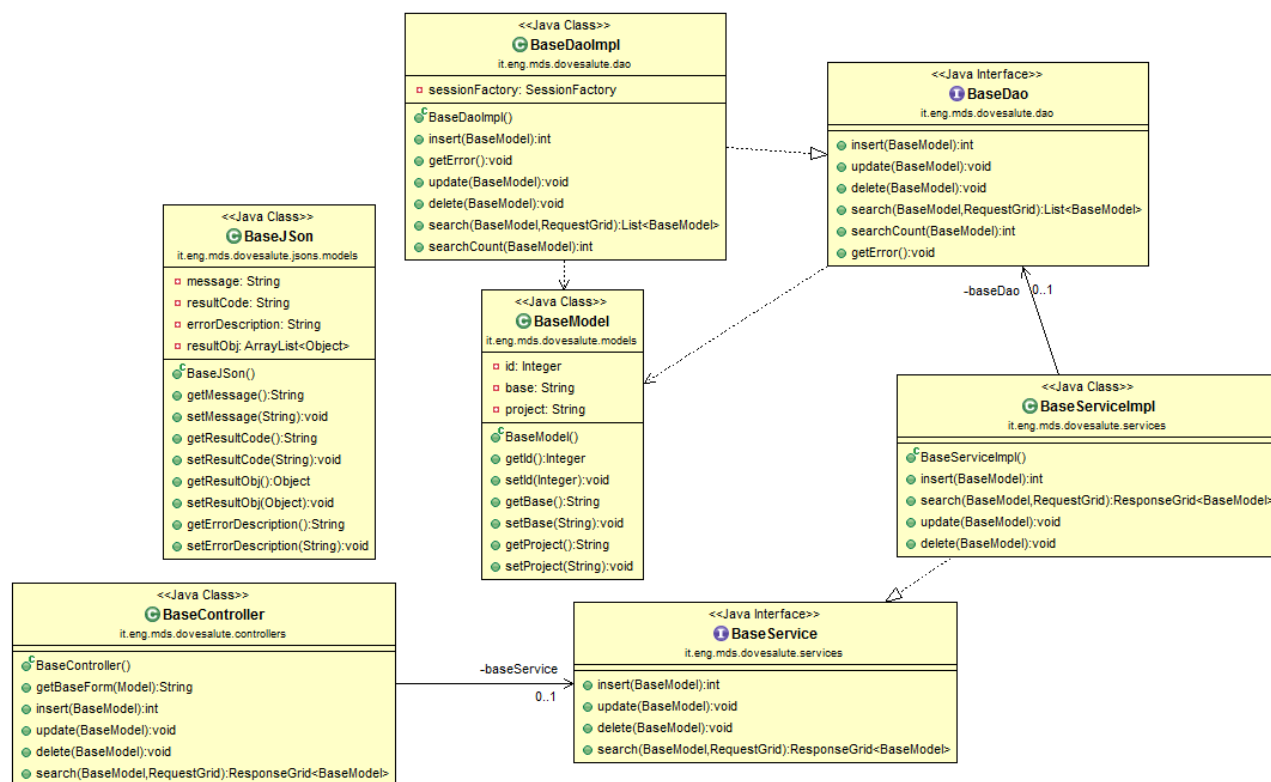


Figura 8 –Class Diagram Generale

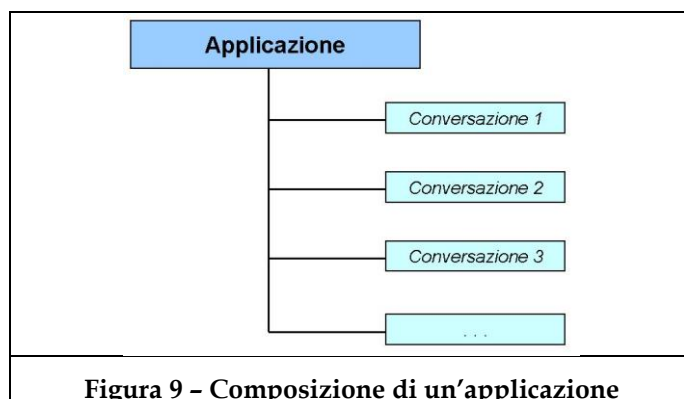
Nome Classe	Descrizione
Base Controller	E' la classe di riferimento per l'implementazione di un Controller.
Generic Action	Interfaccia per l'implementazione di tutti gli oggetti di Service
Base Action Impl	E' la classe di riferimento per l'implementazione di un Service. Implementa l'interfaccia Base Service.
Base Dao	Interfaccia per l'implementazione di tutti gli oggetti DAO
Base Dao Impl	E' la classe di riferimento per l'implementazione di un DAO. Implementa l'interfaccia Base Dao.
Base Model	E' la classe di riferimento che serve per effettuare il mapping tra gli attributi di un oggetto Oracle e una classe bean.
Base Json	E' la classe di riferimento per l'implementazione di un Model Bean contenente la struttura JSON da restituire alla view che effettuerà il rendering delle informazioni da visualizzare all'utente.

3.3 Architettura del framework NSIS

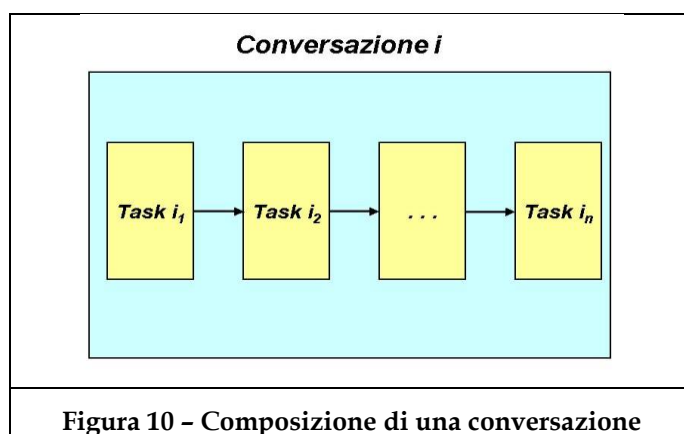
3.3.1 Concetti di base

I concetti basilari che il modello implementativo del framework SISN definisce sono:

- Un'*Applicazione*, dal punto di vista dell'utente, appare composta da un insieme di funzionalità, che chiamiamo *Conversazioni*, il termine mette in evidenza la natura interattiva delle funzioni e immagina l'utente come se stesse "dialogando" con l'applicazione.

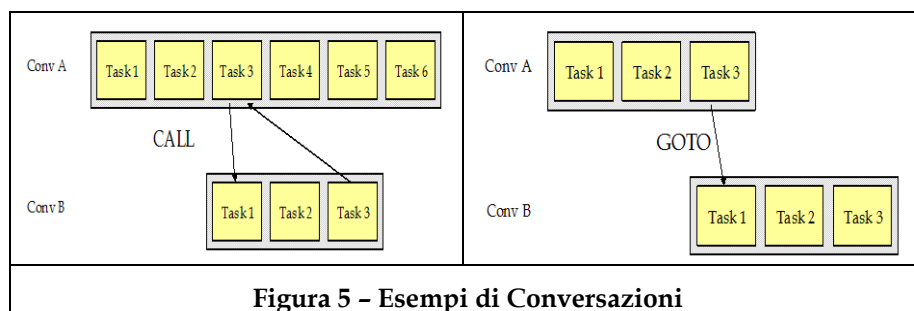


- Una *Conversazione* è un insieme di passi correlati che, assieme, definiscono una funzionalità dell'applicazione. Ogni passo di una conversazione è costituito da una *Pagina* ovvero da una schermata di interazione utente composta da un insieme di oggetti di interazione (*widgets*). L'istanza di una pagina nell'ambito di una specifica conversazione è denominata *Task*. La transizione tra task di una conversazione avviene a mezzo di eventi utente.

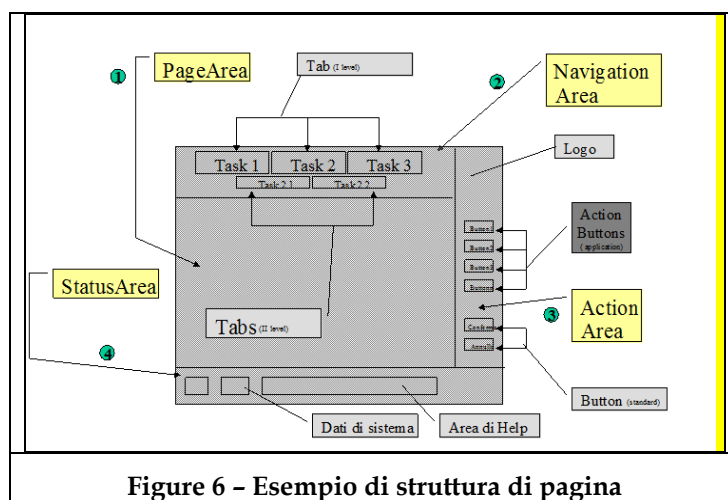


È possibile l'interazione tra conversazioni differenti secondo due modalità:

- **CALL:** da un task di una conversazione è possibile richiamare una nuova conversazione al termine della quale il controllo ritorna al task chiamante
- **GOTO:** da un task di una conversazione è possibile cedere il controllo ad una nuova conversazione.



- Le **Pagine** sono fisicamente implementate da *pagine HTML* (**unità** visualizzata sul client) e dalla corrispondente logica di coordinamento sul server.



- Le *widgets* sono componenti elementari di interazione con l'utente utilizzati nel disegno delle pagine componenti la conversazione. Alcuni esempi di widgets sono i campi per l'immissione del testo (*textfield*), i *combobox*, le *listview* e i *treeview*.

The screenshot shows a web interface with a light blue background. At the top, there are three tabs: 'Criteri di ricerca' (selected), 'Lista Soggetti', and 'Dettaglio Soggetto'. Below the tabs, there are several input fields for search criteria: 'Cognome:', 'Nome', 'Codice fiscale', 'Codice soggetto', 'Località', 'Agenzia', 'Polizza', and 'Costo Polizza'. Each field has a corresponding text input box. To the right of the 'Località' field, there is a button labeled 'Scegli il Comune'. The entire form is enclosed in a black border.

Figura 7 - Esempio di Widgets

- I *servizi di business* contengono la logica di business e di accesso ai dati. Implementano il modello a servizi, per cui non hanno memoria delle interazioni utente. Sono generalmente, ma non necessariamente, implementati con tecnologia consolidata. I servizi di business possono essere implementati per mezzo di *action* o *module*.
- Una *action* è un oggetto di business che evade per intero la richiesta di un servizio applicativo.
- Un *module* è un oggetto di business che può collaborare con altri module per evadere una richiesta di un servizio; tutti i module che collaborano per lo stesso servizio formano un'unità chiamata "pagina logica".

3.3.2 Architettura del framework SISN

Il framework SISN si integra in un'architettura a tre livelli, utilizza la piattaforma J2EE e si basa sul pattern MVC Model-2. La seguente figura schematizza l'architettura del framework evidenziando la corrispondenza esistente con i livelli della piattaforma J2EE e del modello MVC.

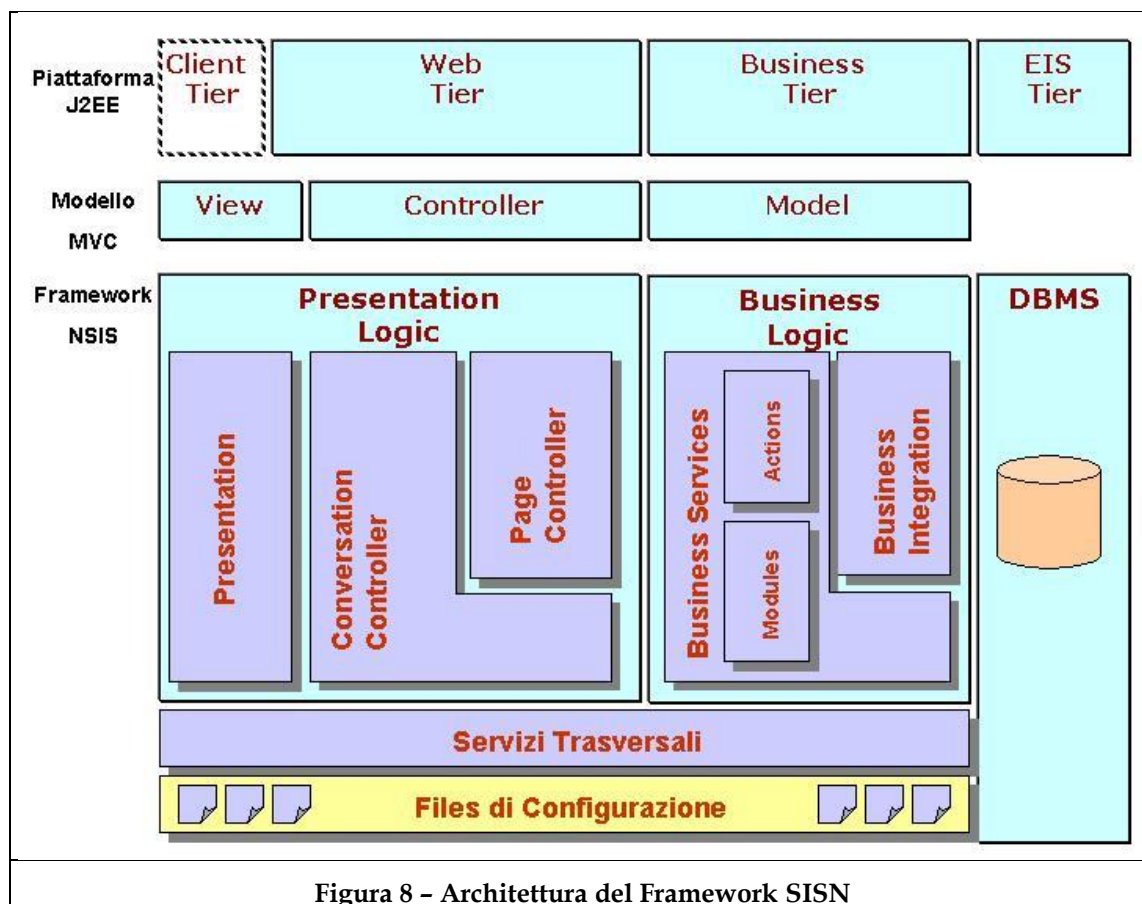


Figura 8 - Architettura del Framework SISN

Nel framework SISN possiamo riconoscere i seguenti componenti:

Sottosistema Transazionale

- **Presentation Logic:** è costituita dall'insieme di componenti che si occupano di gestire l'interfaccia utente;
- **Business Logic:** è la componente del framework che implementa le funzionalità di un'applicazione;
- **Servizi Trasversali:** è la raccolta di servizi comuni e riusabili da tutti componenti del framework;
- **Files di Configurazione:** insieme di file le cui informazioni servono a configurare determinate caratteristiche del framework.

Sottosistema Dati Gestionali

- **DBMS:** è costituito per la quasi totalità da DBMS Oracle ove necessario in configurazione di load balancing attraverso la componente Oracle RAC. In alcune realtà residuali o per pacchetti applicativi esterni i dati sono gestiti con Microsoft SQLServer.

Tutti questi componenti sono in seguito descritti in maniera più dettagliata.

3.3.3 Transazionale - Presentation Logic

Nel modello proposto, le pagine applicative sono implementate come files HTML generati dinamicamente tramite Java Server Pages (JSP).

Nelle pagine JSP non è presente alcuna logica di tipo applicativo. L'unica funzione di tali pagine è di ricevere i dati da visualizzare ed effettuare il *rendering* della corrispondente pagina HTML.

Le pagine JSP sono costruite modularmente a partire da elementi visuali definiti dall'architettura (widgets) attraverso l'utilizzo di uno strumento di disegno opportunamente esteso.

La logica di controllo del flusso applicativo è gestita a livello centrale da un framework architetturale (il cui entry point è il componente *Dispatcher*) all'interno del quale vengono eseguiti i componenti applicativi corrispondenti a ciascuna pagina (i *Page Controllers*) e a ciascuna conversazione (i *Conversation Controllers*).

Le regole di transizione di pagina sono censite in apposite tabelle architetturali (*Conversation Control Table*) che sono accedute dal *Dispatcher*.

Il sottolivello *Presentation* implementa il **View** del modello MVC, mentre il *Conversation Controller* e il *Page Controller* implementano il **Controller**.

3.3.4 Transazionale - Business Logic

La componente dei servizi di business (*Business Services*) realizza le funzionalità di business attraverso due meccanismi alternativi tra loro: le **action** e i **module**.

3.3.4.1 Action

Oggetti di business di questo tipo evadono per intero la richiesta di un servizio all'applicazione; in pratica esiste una corrispondenza univoca fra servizio ed oggetto di business (lo stesso oggetto di business può invece essere usato per evadere richieste diverse).

Ogni *action* è inoltre caratterizzata da uno *scope*, indicando con questo termine l'ambito di vita dell'oggetto, che può avere i seguenti valori:

- **Request** - viene istanziata una nuova *action* ad ogni nuova richiesta;

- **Session** – per tutte le richieste appartenenti alla stessa conversazione (sessione) il servizio viene evaso dalla stessa action;
- **Application** – per tutte le richieste indirizzate al medesimo *container* (JVM) il servizio viene evaso dalla stessa *action*. E' importante sottolineare che un oggetto di business con tale scope non rappresenta un vero *singleton*, in quanto ne esiste una diversa istanza per ogni JVM come ad esempio nei nodi di un *cluster*.

L'esempio seguente mostra il modo con cui viene dichiarata una action all'interno del framework:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<ACTIONS>
<LISTA_UTENTI_ACTION
class="mds.framework.prototype.ListaUtentiAction" scope="REQUEST"/>
</ACTIONS>
```

3.3.4.2 Module

Ogni *module* è un oggetto di *business* che può collaborare con altri *module* per evadere la richiesta di un servizio. Tutti i *module* che collaborano per lo stesso servizio formano un'unità logica chiamata "page logica".

La risposta al servizio invocato è rappresentata dall'unione delle risposte di tutti i *module*. In quale ordine e sotto quali condizioni i *module* che formano una "page logica" vengono invocati è descritto attraverso un semplice *workflow* di esecuzione della logica applicativa. Al termine dell'esecuzione di ogni *module* vengono individuati i successivi *module* da invocare e i parametri di richiesta. L'insieme delle regole che verifica se esiste una transizione fra un *module* ed un altro viene chiamato *conditions*, mentre l'insieme delle regole che determinano i parametri di richiesta di un *module* viene chiamato *consequences*. Prende il nome di *dependencie* invece la relazione per cui successivamente all'esecuzione di un *module* sorgente si verifica un dato insieme di *conditions* e viene invocato un *module* destinatario con i parametri di richiesta recuperati in base ad un dato insieme di *consequences*. L'esecuzione della logica di *business* secondo questa modalità, se da un lato richiede un maggior numero di informazioni da censire, dall'altro consente di identificare degli oggetti con precise responsabilità e riutilizzabili nell'esecuzione di più servizi.

Nell'esempio che segue è riportata la classica situazione di una "pagina logica" che contiene una lista e un dettaglio a fronte di un elemento selezionato.

Quindi, dal punto di vista logico, la "pagina logica" è composta di due moduli, nell'esempio chiamati "AutomaticListaUtenti" e "AutomaticDettaglioUtente", implementati da due hook.

File XML di configurazione delle "pagine logiche"

```
<?xml version="1.0" encoding="ISO-8859-2"?>
<PAGES>
<PAGE name="AutomaticPaginaUtenti">
<PUBLISHER name="/jsp/prototype/listdetail/PaginaUtenti.jsp"/>
<PUBLISHING_MODE name="FORWARD"/>
<PAGE_MESSAGES>
```

```
<IF-MESSAGE name="BEGIN">
<THEN-MODULE name="AutomaticListaUtenti"/>
</IF-MESSAGE>
</PAGE_MESSAGES>
<MODULES>
<MODULE type="custom" name="AutomaticListaUtenti" keep="false"/>
<MODULE type="custom" name="AutomaticDettaglioUtente" keep="false"/>
</MODULES>
</PAGE>
</PAGES>
```

File XML di configurazione dei moduli

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<MODULES>
<DEFAULTS/>
<CUSTOMS
AutomaticListaUtenti="mds.framework.prototype.ListaUtentiModule"
AutomaticDettaglioUtente="mds.framework.prototype.DettaglioUtenteModule"/>
</MODULES>
```

L'implementazione "a moduli" permette inoltre di usufruire di due moduli standard (DefaultListModule e DefaultDetailModule) presenti nel framework che, tramite files di configurazione (moduli, liste, dettagli, query), permettono di ottenere una pagina logica composta da una lista e un dettaglio (comprensivo di select/insert/update) senza scrivere del codice Java.

3.3.4.3 Componenti di business logic e flusso di esecuzione

Nella figura seguente è mostrato un esempio di flusso di conversazione che prevede l'interazione con due moduli di business logic, rispettivamente in modalità action e module.

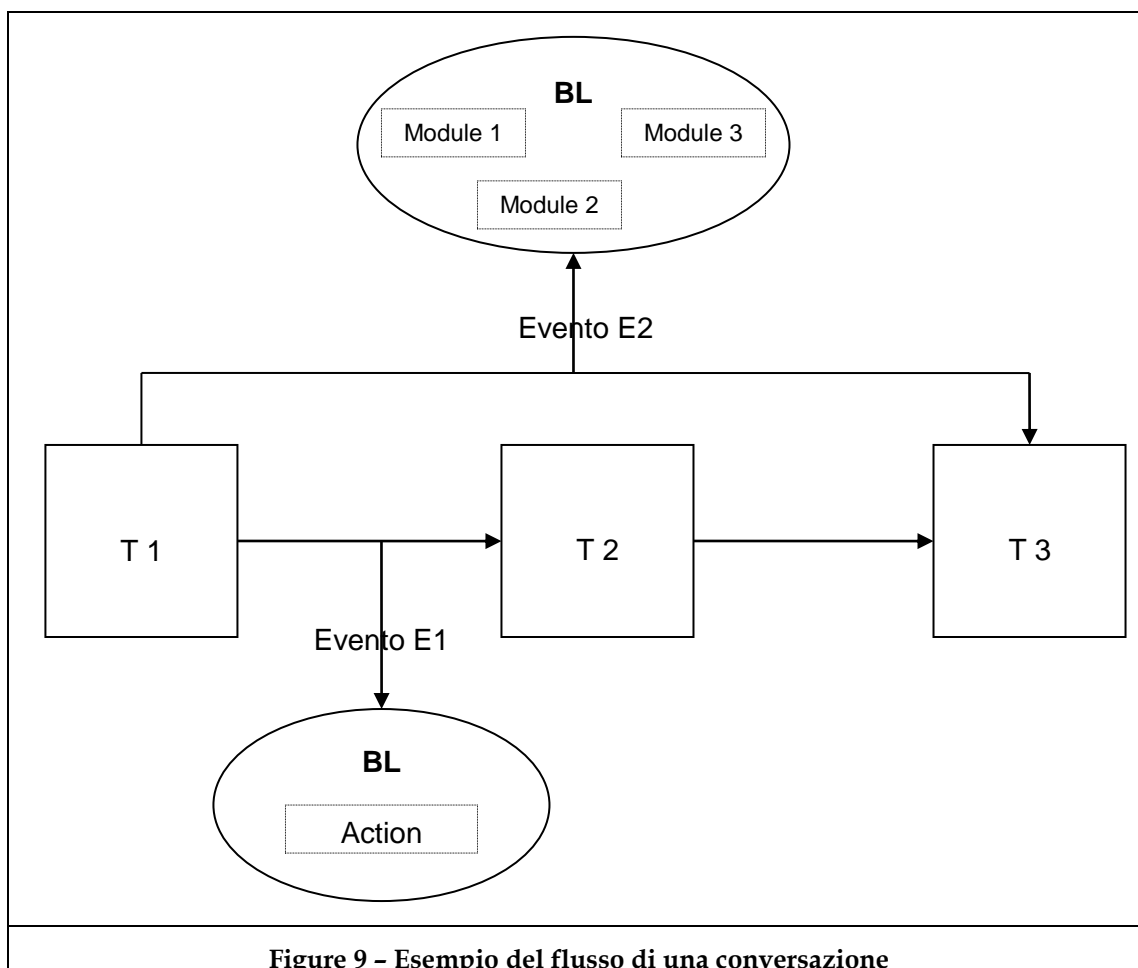


Figure 9 - Esempio del flusso di una conversazione

Nell'esempio è descritta una conversazione costituita da tre task e dalle relazioni tra di essi. Dal task T1, in seguito all'evento di interazione E1, è richiamata una funzione di business logic in modalità action, quindi il sistema passa al task T2. L'evento E2 permette il passaggio dell'applicazione dal task T1 al task T3 a seguito dell'invocazione di un servizio di business logic in modalità module.

La figura seguente mostra nel dettaglio i componenti architetturali coinvolti durante l'invocazione di un servizio di business logic nelle due modalità action e module.

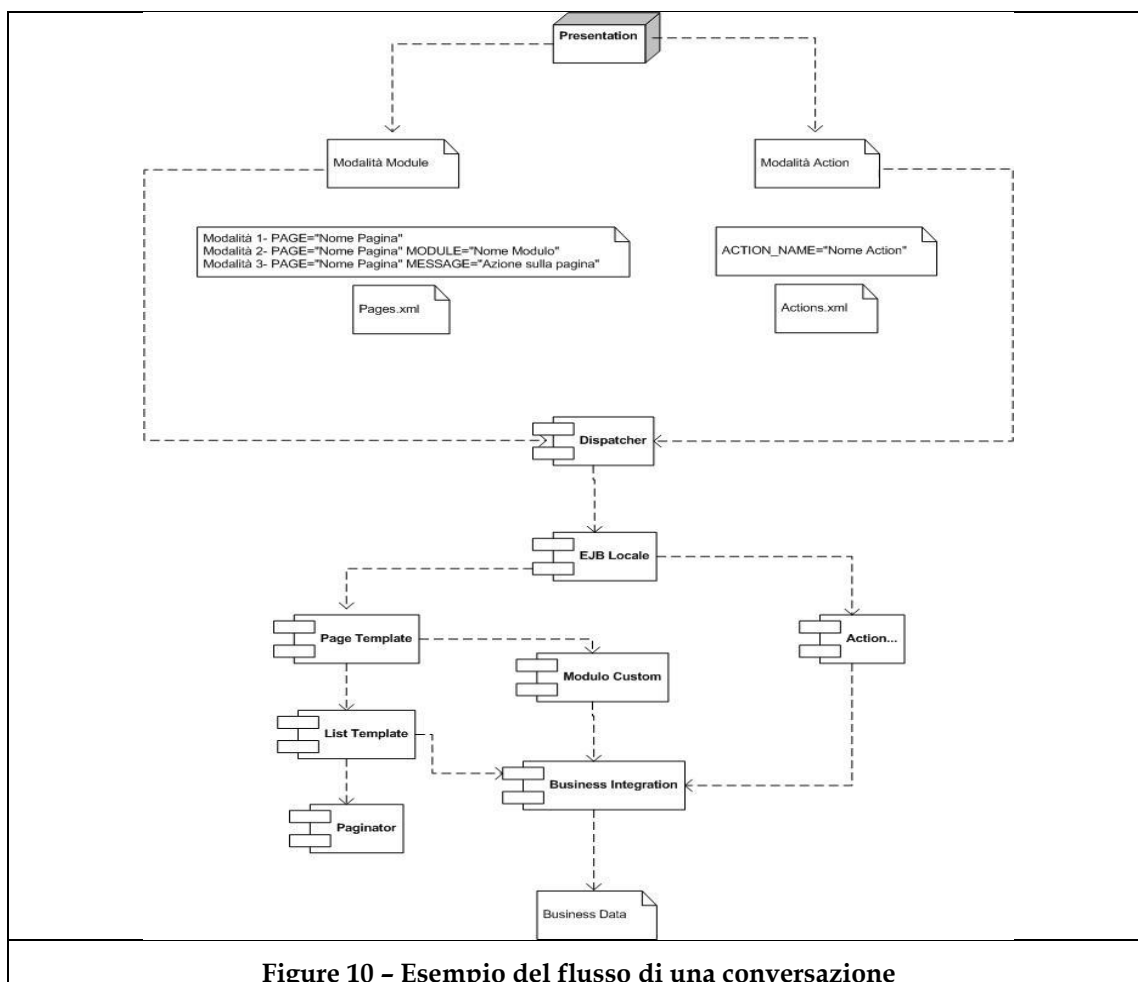


Figure 10 – Esempio del flusso di una conversazione

- **Dispatcher:** ha il compito di gestire la navigazione tra i diversi componenti dell'applicazione;
- **EJB Locale:** è il componente che attiva la logica di business in base alla modalità richiesta;
- **Action:** esegue una funzionalità di business logic in modalità action;
- **Module**
 - **Page Template:** esegue i moduli previsti in una pagina logica ;
 - **Module Custom:** è un modulo che contiene la business logic;
 - **List Template:** è un modulo standard del framework che implementa una lista;
 - **Paginator:** è un modulo standard del framework che implementa i meccanismi di paginazione della lista (pagina avanti, pagina indietro, ...etc);
- **Business Integration:** si occupa dell'interfaccia verso il DBMS;
- **Business Data:** è costituito dal DBMS.

3.3.4.4 Business Integration

Questo modulo virtualizza l'accesso ai dati: aggiunge delle funzionalità a quelle già presenti in JDBC (che è già un layer che virtualizza l'accesso ai dati). L'esigenza che si prefigge di risolvere il

modulo di accesso ai dati è quella di minimizzare lo sforzo richiesto per migrare un applicativo sviluppato facendo uso di un database, su un altro database.

Utilizzando JDBC questa migrazione richiederebbe molto lavoro; il Business Integration, invece, astrae da JDBC e indipendentemente dal database utilizzato fornisce una serie di funzionalità come ad esempio gli *scrollable result set*, la gestione dei *pool di connessione*, etc.

Questo sottosistema offre le potenzialità di JDBC2 su qualunque tipo di database, permettendo la migrazione delle applicazioni su vari database, con uno sforzo minimo. Il modulo di accesso ai dati è utilizzabile sia da *action* che da *moduli*.

Nella progettazione del modulo di accesso ai dati si è fatto molto uso di **Design Pattern**, in particolare:

- l'oggetto `DataConnectionManager` è stato implementato come un Singleton
- si sono utilizzati i pattern **AbstractFactory** e **FactoryMethod** per la creazione di tutti quegli oggetti che debbano avere un'implementazione differente pur mantenendo una stessa interfaccia.
Gli oggetti di tipo `DataField`, `ScrollableDataResult`, `SQLCommand` sono esempi di oggetti per la quale la creazione passa attraverso un `AbstractFactory` o un `FactoryMethod`.
- Nel caso degli oggetti di tipo `SQLCommand` l'oggetto che funge da `AbstractFactory` configura degli oggetti conformi al pattern **Command**.
- Gli oggetti di tipo **ConnectionPoolInterface** gestiti dall'oggetto `DataConnectionManager` vengono creati attraverso una factory che lavora con l'ausilio dell'introspezione (questo per evitare in fase di compilazione di dover includere i jar di tutti i vendor)

3.3.5 Transazionale - Servizi trasversali

Di seguito sono descritti i servizi e i componenti trasversali a tutti i moduli.

- **Tracer** – è utilizzato per la memorizzazione su *file* dell'attività applicativa; ad ogni messaggio viene inoltre assegnata una *severity* che ne discrimina l'effettiva memorizzazione a seconda del livello minimo di *tracing* configurato;
- **Error Handler** – ha la responsabilità di mantenere uno *stack* degli errori applicativi e non; ad ogni errore viene inoltre assegnata una *severity*; due tipologie di errori sono state identificate al momento:
 - **User Error** – errori riscontrati nella logica applicativa ed identificati da un codice; tale codice viene risolto a *runtime* con una descrizione catalogata dell'errore.
 - **Internal Error** – errori generati da sottosistemi terze parti (quali ad. es. l'`SQLException` di un attività JDBC).

- ***Request Container*** – ha la responsabilità di mantenere le informazioni il cui contesto è la specifica richiesta di un servizio (per il canale HTTP ha la stessa vita dell'oggetto *HttpServletRequest*);
- ***Session Container*** – ha la responsabilità di mantenere le informazioni il cui contesto è la conversazione del *device* con l'applicativo (per il canale HTTP ha la stessa vita dell'oggetto *HttpSession*);
- ***Application Container*** – ha la responsabilità di mantenere le informazioni il cui contesto è l'applicativo stesso; tale componente trasversale alle conversazioni con i *device* assume fondamentalmente il significato di una *cache* (è infatti possibile specificare un tempo di scadenza degli oggetti in esso mantenuti);
- ***Configuration*** – ha la responsabilità di mantenere i dati di configurazione dell'applicativo censiti su *file XML*;
- ***InitializerManager*** – ha la responsabilità di coordinare le attività di inizializzazione dell'applicativo nella fase di *bootstrap*;
- ***Navigator*** – ha la responsabilità di mantenere lo *stack* delle richieste dei servizi evasi consentendo la riesecuzione di un precedente servizio nelle medesime condizioni di *Request Container* e *Session Container*. Consente ad. es. di sostituire il *refresh* ed il *back* di un *browser* con una gestione applicativa e più sofisticata delle due funzionalità.

3.3.6 Transazionale - Files di configurazione

3.3.6.1 Presentation Logic

- ***Properties file applicativi*** – E' possibile definire per ogni pagina dei *properties file* applicativi. Questi files devono risiedere in una struttura di *directory* con nome corrispondente all'area funzionale della conversazione a cui fanno riferimento. Inoltre i *properties files* devono avere il nome del controller da cui saranno acceduti, ed estensione *.properties*;
- ***File di configurazione di pagina*** – Tale file, con nome corrispondente a quello della pagina ed estensione *DCC*, rappresenta il componente che contiene la descrizione a Design Time della Pagina (tutti i componenti in essa presenti con i valori iniziali delle proprietà). Esso consente di inizializzare il contenuto della pagina da visualizzare prima del rendering;
- ***File di configurazione di Conversazione (Conversation Control table)*** – Tale file, che ha nome corrispondente a quello della conversazione ed estensione *CCT*, è un documento XML. Contiene l'insieme delle informazioni necessarie a descrivere il flusso conversazionale e la UI delle parti di generazione degli eventi utente (componenti di navigazione e di invocazione della business logic).

3.3.6.2 Business Logic

La configurazione della business logic è controllata da un unico file chiamato *master.xml*, all'interno del quale sono memorizzati i percorsi relativi dei file di configurazione da caricare.

Mediante l'oggetto singleton *ConfigSingleton* è possibile accedere a tutte le informazioni presenti in essi. I file di configurazione utilizzati dalla business logic sono i seguenti:

- **actions.xml**
- business_map.xml
- common.xml
- **data_access.xml**
- dispatchers.xml
- initializers.xml
- master.xml
- **modules.xml**
- **pages.xml**
- proxies.xml
- security.xml
- **statements.xml**
- tracing.xml

Nel caso in cui venga creato un nuovo file di configurazione, è sufficiente registrarlo nel file *master.xml* per avere a disposizione le informazioni tramite il *ConfigSingleton*.

Nella lista sono stati posti in evidenza (in grassetto) i files .xml che necessitano di una configurazione da parte dello sviluppatore; gli altri file contengono dei valori di default che possono essere modificati per estendere le funzionalità del framework.

3.3.7 Dati Gestionali - DBMS

Il DBMS **costituisce l'infrastruttura dati fondamentale per il supporto dei sistemi che prevedono una interazione diretta con gli utenti** (ad esempio i sistemi transazionali). Il SISN fa principalmente uso di DBMS Oracle, usati dai sistemi transazionali per permettere la conservazione delle informazioni gestite dagli utenti attraverso le applicazioni. Vista l'eterogeneità del sistema SISN, il DBMS è il contenitore di **molteplici tipologie d'informazione**, indipendentemente:

- dell'area funzionale cui l'applicazione appartiene,
- dell'area organizzativa degli utenti che trattano le informazioni attraverso le applicazioni (ASL, utenti Ministeriali, utenti della Regione, ...),
- della tipologia di dato trattato (dati elementari, documenti, ...).

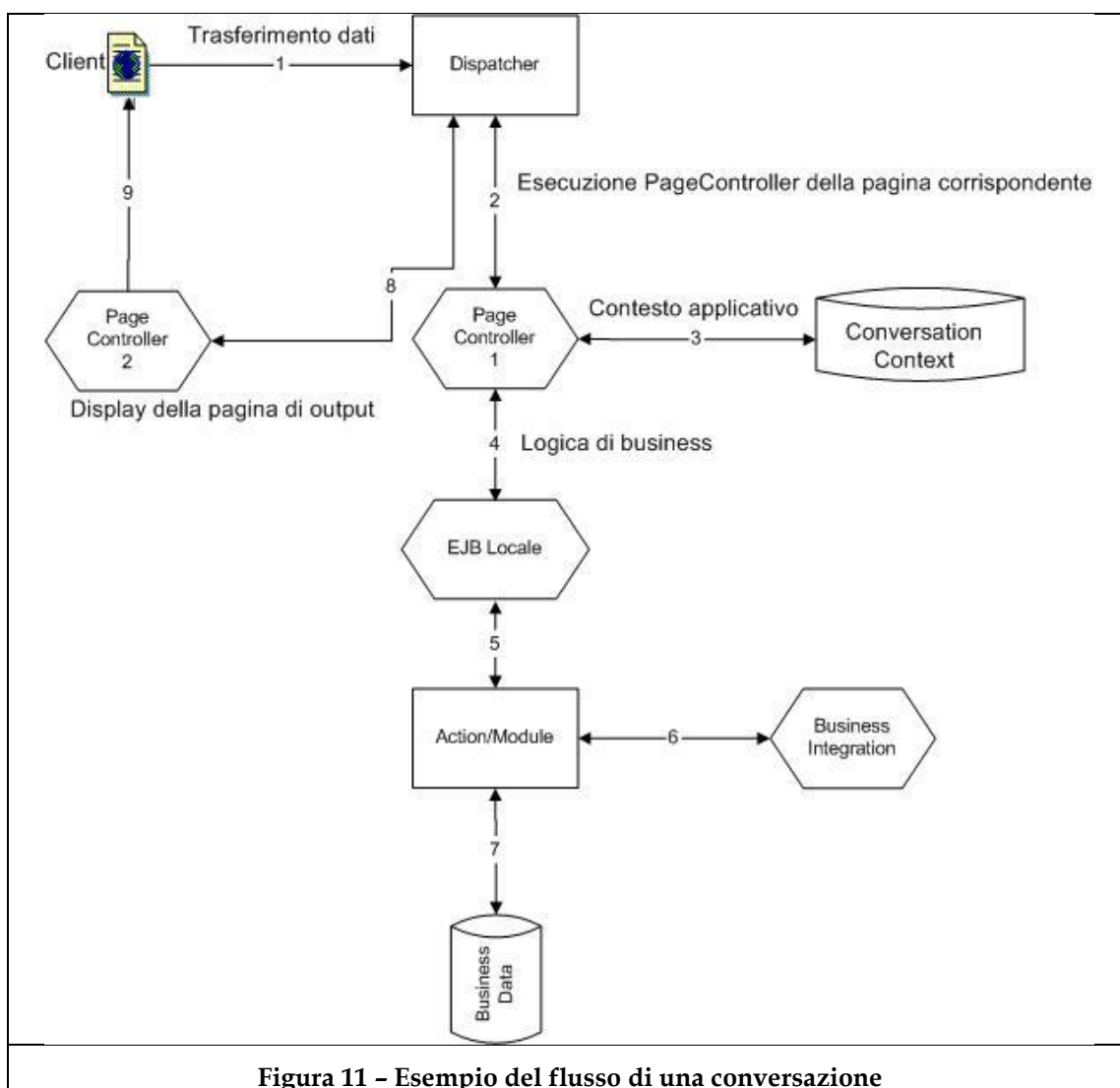
Tipicamente il DBMS è alimentato dalle applicazioni del sottosistema transazionale, ma vi sono casi di inserimento dei dati attraverso procedure batch (generalmente per inserimenti massivi), o caricamenti di dati posticipati rispetto all'inserimento di questi da parte dell'utente (ad esempio per dati che necessitano di una successiva validazione da parte di un responsabile). Il DBMS ha le seguenti principali caratteristiche:

- ambiente di consuntivazione dei dati base (eventi elementari) su logiche relazionali.
- elevata granularità del dato;
- tecnologia relazionale orientato alla gestione dei soli dati utili per il corretto funzionamento delle applicazioni (i dati non più necessari passano attraverso un processo di storicizzazione);
- costituisce il centro di raccolta dei dati elementari che saranno successivamente trattati e aggregati per successive analisi (ad esempio attraverso gli strumenti messi a disposizione dal sottosistema di Datawarehousing);
- la struttura logica è in genere complessa, al fine di dare supporto per rispondere a tutti i requisiti delle applicazioni;
- garantisce la completa tracciabilità e ricostruibilità delle informazioni inserite, sia da un punto di vista temporale (quando l'informazione è stata inserita), sia da un punto di vista di responsabilità (chi ha inserito l'informazione).

Tutti i dati resi disponibili dal DBMS vengono mappati e gestiti attraverso operazioni di inserimento, modifica e storicizzazione eseguite direttamente dagli utenti responsabili del trattamento del singolo dato.

3.3.8 Esempio di interazione utente-framework

Nella figura seguente sono illustrati i vari componenti del framework coinvolti nel corso di una completa interazione dell'utente con l'applicazione in seguito alla richiesta di un servizio applicativo.



1. L'utente richiede un servizio applicativo. In seguito all'evento generato il browser invoca il dispatcher del framework inviandogli le informazioni necessarie all'esecuzione della funzionalità di business.
2. Il dispatcher invoca il controller della pagina da cui ha avuto origine la richiesta.
3. Il controller di pagina richiama i dati di contesto della pagina (credenziali dell'utente connesso, ...).

4. Il controller di pagina invoca l'EJB locale indicando la funzionalità di business richiesta, la modalità di esecuzione e gli argomenti relativi.
5. L'EJB locale invoca la funzionalità di business richiesta in base alla modalità (action o module).
6. La funzionalità di business invoca la business integration al fine di gestire le interrogazioni verso il DBMS.
7. La funzionalità di business interagisce con il DBMS e successivamente il controllo torna al dispatcher.
8. Il dispatcher invoca il page controller della nuova pagina da visualizzare.
9. Il page controller costruisce la pagina HTML di risposta al servizio applicativo richiesto dall'utente e la invia al browser.

4 Architettura di Sviluppo

In questo capitolo vengono descritti gli strumenti, integrati nel framework SISN, attraverso i quali è possibile gestire le modifiche evolutive delle applicazioni attualmente in uso e alcuni accenni sui framework di nuova generazione, che consentono un rapido e sicuro sviluppo, utilizzati per l'implementazione delle nuove applicazioni.

Lo sviluppo applicativo, che segue la fase di analisi funzionale nella quale vengono descritte, tramite notazione UML, le funzionalità da implementare nel sistema (Use Cases e Sequence Diagram), si svolge attraverso le seguenti fasi:

- **Disegno tecnico** – vengono aggiunte al modello le informazioni di dettaglio necessarie allo sviluppo applicativo, in particolare per il supporto alla generazione automatica (Navigation Diagram, Page Controller);
- **Programmazione** – vengono sviluppate le singole componenti applicative: ogni area richiede skill specifici (Pagine, Page/Conversation Controller, EJBs, Connectors).

Per l'analisi funzionale e tecnica si utilizza il formalismo UML (Unified Modeling Language) per rappresentare le applicazioni rispetto al modello individuato, per i seguenti motivi:

- è una notazione standard universalmente riconosciuta;
- permette di dare una chiara definizione delle interfacce tra i sottosistemi;
- descrive in modo omogeneo le strutture dati e le relative operazioni.

4.1 Modellazione delle Conversazioni

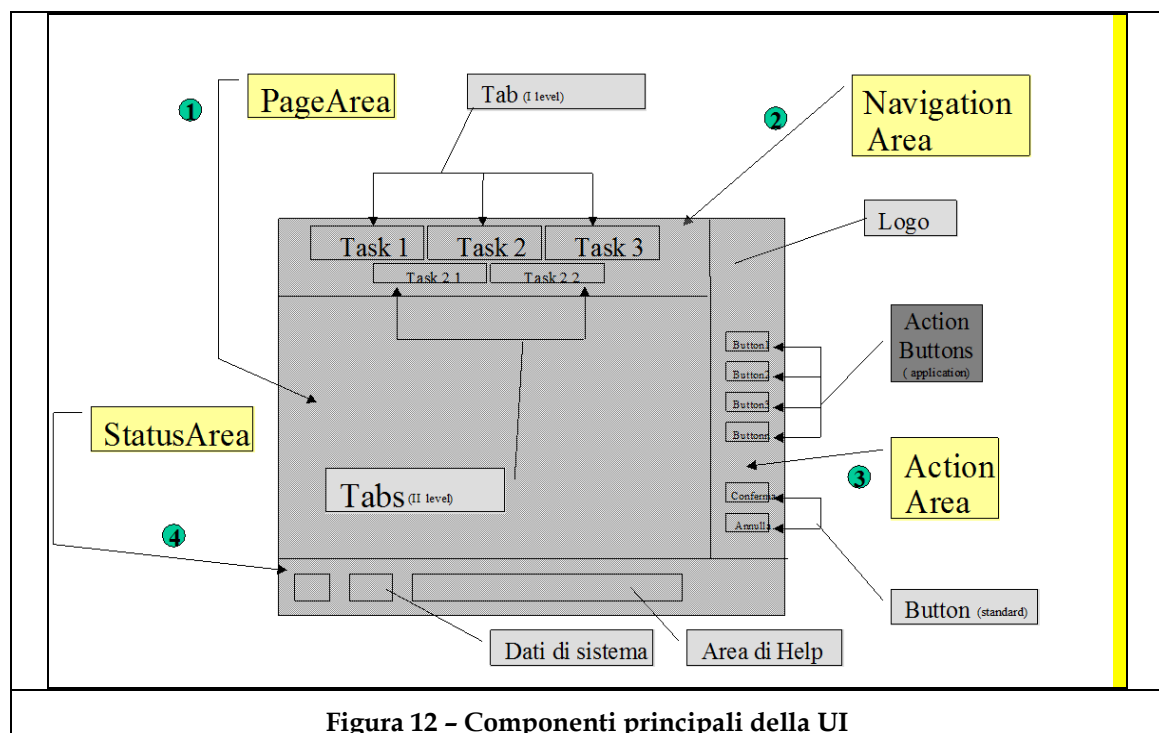
Nel modello applicativo del framework:

- le *Pagine* (classi) sono utilizzate come *Tasks* (istanze di pagina) nelle conversazioni;
- i *Navigation diagrams* sono utilizzati per esplicitare le azioni utente (user actions) e le corrispondenti transizioni di pagina;
- le Pagine possono essere riutilizzate in più Tasks;
- i task possono avere eredità multipla.

4.2 Interazione utente

4.2.1 Elementi e semantica dell'Interfaccia Utente

Le componenti principali della *Interfaccia Utente* (UI) di una applicazione sono indicate nella figura seguente:



In essa:

1. **PageArea** - è la pagina applicativa correntemente visualizzata, relativa al Task corrente
2. **NavigationArea** - è la sezione dello schermo che contiene gli elementi di UI che governano la navigazione di pagina, che contiene i *Tab di navigazione*.
 - **Tab** - rappresenta il singolo elemento di navigazione; per ogni *Task* raggiungibile da una conversazione esiste un *tabselezionabile*

Si definiscono due livelli di *Tab*:

- **Main** - consente la navigazione tra i Tasks principali della conversazione
- **Secondary** - consente la navigazione tra i Tasks costituenti una sezione principale

Un *Tab* possiede i seguenti *attributi di visualizzazione*:

- **Visibilità**- Quando falso, il Task corrispondente non è raggiungibile dallo stato di navigazione corrente (il *tab* non è visualizzato)
- **Abilitazione**- Quando falso, il Task corrispondente non è raggiungibile dallo stato di navigazione corrente (il colore del *tab* è grigio)
- **Stato applicativo** - Può assumere i seguenti valori:
 - **Normal** - il Task corrispondente non è stato mai raggiunto e non è quello corrente
 - **Current** - il Task corrispondente è quello correntemente visualizzato (il colore del *tab* risulta più marcato di quello dello stato di *Normal*)
 - **Correct** - il Task corrispondente è stato eseguito in una fase di navigazione passata ed i dati inseriti hanno superato le validazioni e si è consentito di abbandonarlo (il *tab* contiene un simbolo di spunta ✓ di colore verde)
 - **Error** - il Task corrispondente è stato visualizzato in una fase di navigazione passata ed i dati inseriti non hanno superato le validazioni (il *tab* contiene un simbolo ✗ di colore rosso)
 - **Incomplete**- il Task corrispondente è stato visualizzato in una fase di navigazione passata ed i dati inseriti non sono completi (il *tab* contiene un simbolo 📅 di colore giallo)

3. **ActionArea**- è la sezione dello schermo che contiene gli elementi di UI che permettono l'applicazione di azioni ai dati rappresentati nel Task corrente. Una azione può provocare una transizione di Task (anche verso lo stesso Task) oppure una passaggio di conversazione. La *ActionArea* tipicamente contiene dei *Button*.

- **Button** - rappresenta il singolo elemento di generazione degli eventi di azione

Esistono due tipologie di *Button*:

- **Application** - sono i *Button* specifici dell'applicazione
- **Standard** - sono presenti in tutte le applicazioni e comprendono:
 - **Conferma** - Conferma le operazioni effettuate durante la esecuzione della conversazione
 - **Annulla** - Annulla le operazioni effettuate durante la esecuzione della conversazione. Fa ritornare alla conversazione invocante

Un *Button* può assumere i seguenti *stati di visualizzazione*:

- **Enabled** - il *Button* è visibile e selezionabile: può generare l'evento corrispondente
- **Disabled** - il *Button* è visibile ma non è selezionabile
- **Invisible** - il *Button* non è visibile

4. **StatusArea**– è la sezione dello schermo che contiene elementi informativi sul sistema sull'applicazione in esecuzione:

- **Logo** – indica il logo della compagnia e dell'area applicativa in esecuzione
- **Area di Sistema** – indica le informazioni legate al sistema e all'utente (Data, User Name)
- **Area di Help** – è la sezione dello schermo che contiene un'aiuto generico o contestuale alla pagina o al campo, nella forma di due Buttons:
 - **Manuale** – Invoca in una finestra separata il manuale utente
 - **Cosa Fare** – Invoca in una finestra separata il manuale utente nella sezione che descrive il Task corrente

4.2.2 Eventi

L'evoluzione del flusso conversazionale è governata dagli *eventi* che l'utente dell'applicazione genera interagendo con gli elementi di interfaccia (*Tab* e *Button*) e dalla risposta applicativa agli eventi utente.

Gli eventi utente vengono classificati nel seguente modo:

- **UserEvent**– Generati direttamente dall'utente, si suddividono in:
 - *NavigationEvent*: evento scatenato selezionando un *Tab* della *NavigationArea*
 - *ActionEvent*: evento scatenato selezionando un *Button* della *ActionArea*
- **ControllerEvent**: esito della elaborazione dell'evento utente (*UserEvent*) durante la fase *Display* (visualizzazione di una nuova pagina) o *Process* (invocazione di una funzionalità di business logic) di un *Page Controller*

La navigazione applicativa è gestita, in risposta a *ControllerEvents*, come specificato dalla tabella di gestione della conversazione (*Conversation Control Table*).

Il *Page Controller* ha la possibilità di influire sul flusso di conversazione ritornando appropriati *ControllerEvents*, e segnalando eventuali errori all'architettura tramite un'appropriata *API*.

4.3 Gestione del contesto applicativo

Nell'architettura delineata, si possono evidenziare diversi livelli di scambio dati (contesti) tra i vari componenti, in particolare:

- **Contesto di Conversazione (Conversation Context)**- Immagazzina lo stato della conversazione corrente ed è accessibile a tutti i Page Controllers che appartengono alla *conversazione* corrente e al Conversation Controller della Conversazione. E' suddiviso tra:
 - **Current** - contesto della conversazione corrente, e
 - **Parent** - contesto della conversazione chiamante
- **Contesto Display (DisplayContext)**- Contiene oggetti (widgets) che rappresentano l'immagine dei campi di user interface presenti sulla pagina correntemente visualizzata. Questi oggetti sono automaticamente resi disponibili dall'architettura, e rappresentano il metodo di interazione tra il PageController e i campi visuali della pagina corrispondente.