STANDARD PROGRAMMAZIONE

APPLICAZIONI WEB JAVA J2EE (JAVA 2 ENTERPRISE EDITION)

Ver. 3.0

TABELLA DELLE VERSIONI

Versione	Data	Descrizione delle modifiche
1.0	Giugno 2001	Prima stesura, inserita in Standard Parte 3 - programmazione
2.0	Maggio 2002	Trattazione convenzioni standard di sviluppo, inserita in Standard Parte 3 - programmazione
3.0	Aprile 2004	Aggiornamento intero documento, scorporo da Parte 3

INDICE

1 Introduzione

Il presente documento intende definire le convenzioni standard per lo sviluppo di codice Java a cui lo sviluppatore dovrà allinearsi, al fine di garantire un prodotto qualitativamente soddisfacente, rispondente al tempo stesso ai necessari requirement di manutenibilità e portabilità del codice. Il documento è rivolto a personale tecnico che possieda già un buon livello di conoscenza del linguaggio.

La maggior parte delle applicazioni JAVA realizzate per il Ministero dell'Economia e delle Finanze sono oggi in esercizio sull'application server IBM WebSphere release 3.5.4, non conforme alle specifiche Java 2 Enterprise Edition.

Questa piattaforma sarà tuttavia presto migrata verso la release 5.x, aderente alle specifiche sopracitate.

1.1 Scopo del documento

Lo scopo di questo documento è di indicare le linee guida principali, best practices, esempi e metodologie di sviluppo per le applicazioni gestionali "Enterprise" di tipo Web, conformi allo standard J2EE, da sviluppare in ambito MEF.

Saranno indicate le linee guida da seguire durante le varie fasi di sviluppo che vanno dall'analisi dei requisiti funzionali e non funzionali, all'individuazione dei patterns architetturali e conseguente definizione dell'architettura e del micro disegno applicativo, utilizzando eventuali patterns di disegno e testing (unit, integration, system, performance).

1.2 Definizioni ed Acronimi

Codice	Descrizione delle modifiche
AD	Active Directory
API	Application Programming Interface
CCITT	Consultative Committee on International Telephony and Telegrapy
DAP	Directory Access Protocol
DIT	Directory Information Tree
EIS	Enterprise Information Server
GUI	Grafic User Interface
IP	Internet Protocol
ISO	International Standard Organization
J2EE	Java 2 Enterprise Edition
LDAP	Lightweight Directory Access Protocol
LUW	Logical Work Unit
MVC	Model View Controller
OID	Oracle Internet Directory
OSI	Open Systems Interconnection
SOA	Service Oriented Architecture

SOAP	Simple Object Access Protocol
SSO	Single Sign On
TCP	Transmission Control Protocol
UDDI	Universal Descriptor Directory Interface
XML	eXtensible Markup Language
WIA	Windows Integrated Autentication

2 Premessa

Il modello di sviluppo che viene proposto in questo documento intende rispondere ad una serie di requisiti di carattere sia *tecnologico* (client leggero, portabilità, scalabilità, ...), sia *metodologico* (modelli di analisi, disegno, documentazione, ...).

Le applicazioni attualmente in produzione, non conformi alle specifiche J2EE, già adottano un modello computazionale distribuito, che prevede la logica applicativa suddivisa tra diversi "layer" logici.

Nella prima parte del presente documento saranno indicati ulteriori paradigmi per lo sviluppo di codice J2EE:

- utilizzo di Design Patterns;
- sviluppo tramite frameworks;
- integrazione di componenti.

Nella seconda parte saranno invece affrontate problematiche più specificatamente legate alla fase di codifica, fornendo alcune "best practices" di programmazione, relative ad aspetti quali l'organizzazione dei package e regole di naming, il logginig, le modalità di commento del codice.

3 Architettura J2EE

L'Architettura di un'applicazione ha l'obiettivo di definire la struttura di una soluzione ad un problema di business.
Una buona architettura deve soddisfare i seguenti requisiti:

- Essere flessibile ed estendibile per accomodare nuovi requisiti.
- Essere costituita da componenti le cui interfacce siano ben identificate in modo da parallelizzare le fasi di disegno e sviluppo.
- Fornire un framework ed un insieme di guidelines per tutte le scelte hardware, software e rete.

3.1 Architettura di Riferimento

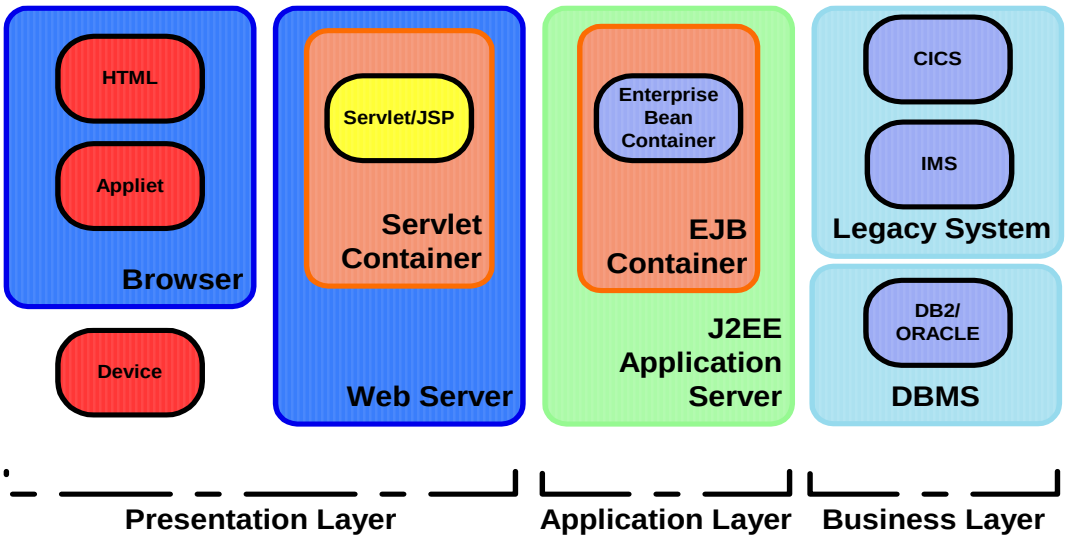
L'architettura di riferimento già in uso presso il MEF prevede un modello computazionale multilivello distribuito, articolato sui segueti "Layer":

- il *Presentation Layer*, costituito dal Client Tier e dalla Client Logic (Web Tier);
- *l'Application and Integration Layer*, costituito dalla Application Logic, dai Business Services Objects, dagli Host Integration Objects e dai Connectors;
- *l'Enterprise Information Layer* (Business Layer), costituito, per esempio, da applicazioni ospitare in un TP Monitor (ad es. CICS) ed un DBMS (DB2 e/o Oracle).

Vediamo in dettaglio lo scopo di ogni layer elencato:

Il presentation layer: ha la responsabilità di indirizzare le problematiche di presentazione e di accesso al mondo applicativo attraverso vari devices: browser, thin device, etc.

L'application layer: riveste un ruolo fondamentale poiché il suo compito è sia quello di fornire la logica di business per le applicazioni che di integrare attraverso una serie di servizi standard quella residente sui tradizionali 'EIS'.



I vantaggi che raccomandano l'utilizzo di un modello di sviluppo basato su più layers sono molteplici:

- **Scomposizione Funzionale**

L'applicazione è suddivisa in componenti ai quali sono affidati ruoli ben precisi, questo contribuisce a strutturate l'analisi, lo sviluppo e la successiva manutenzione.

- **Modularità**

La distinzione di responsabilità tra i vari layers e la definizione precisa delle interfacce tra di loro consente di caratterizzare il disegno in moduli ognuno dei quali indirizza problematiche ben definite: presetazione, logica di business, data acces.

- **Identificazione degli skills**

I moduli che compongono i diversi layers richiedono skill di disegno che di programmazione diversi, la definizione delle interfacce tra layers consente quindi che sia lo sviluppo che il test dei singoli layers possa procedere in modo parallelo e indipendente.

- **Separazione della Presentazione dal modello di Business**

La separazione tra layers consente la possibilità di sviluppo di diverse tipologie di presentazione e di accesso allo stesso modello di business

- **Riuso**

La separazione dei ruoli e la definizione precisa delle interfacce tra moduli consente un facile riutilizzo degli stessi in contesti diversi, migliorando la qualità e consentendo un risparmio sia sui tempi di sviluppo che sui costi

- **Gestione dei Cambiamenti**

Le interfacce definiscono i contratti tra i diversi layer, questo consente la modifica o addirittura la sostituzione dei componenti di un layer senza che ciò obblighi modiche negli altri layers

- **Scalabilita'**

La distinzione logica consente la distribuzione su più livelli fisici scelti in funzione del carico applicativo, delle performance richieste e dei costi di gestione.

- **Diverse Tipologie di Client**

Le diverse tipologie di thin client sono risolte nei primi due layers in modo trasparente alle logiche applicative e di business, questo consente persino l'uso contemporaneo di diverse tecnologie di presentazione.

- **Flessibilità**

Lo schema a quattro livelli rappresenta lo schema più generale. Alcune applicazioni possono richiedere uno schema con un numero inferiore di livelli.

3.2 Patterns Architetturali J2EE

Il modello a layers incoraggia l'adozione dei cosiddetti *Design Patterns*.

Un *Design Pattern* è un modello consolidato (una raccomandazione) per gestire e risolvere i più comuni e ricorrenti problemi di business attraverso il disegno di classi e/o interfacce che interagiscano tra di loro secondo modalità ben precise e consolidate.

Esiste ormai un esteso catalogo di patterns, ciascuno di essi identificato attraverso un nome, il contesto dove è applicabile e il problema che risolve. Dall'adozione di un determinato design pattern segue lo sviluppo di codice Java sotto forma di classi, interfacce, e framework.

In genere un design pattern comprende:

- La descrizione del problema che esso indirizza
- La descrizione della soluzione suggerita, che comprende una serie di diagrammi di classi e diagrammi di sequenza, ed esempi di codice
- La descrizione dei benefici che possono derivare dall'adozione del pattern stesso.

Nel seguito di questo documento saranno brevemente descritti alcuni dei pattern principali, e sarà esplicitamente indicato quali da utilizzare sui vari layers.

3.2.1 Model View Controller

Model-View-Controller (MVC nel seguito) è storicamente il primo pattern sistematicamente codificato (agli inizi degli anni '80), ed evolvendo nel tempo è rimasto alla base della maggior parte dei pattern di programmazione moderni.

Il modello prende spunto dalla considerazione che le applicazioni che presentano commistione di componenti logica di accesso al dato, di logica di business e di logica di presentazione, risultano particolarmente onerose da manutenere, in quanto ogni modifica, anche piccola, si riverbera su numerose componenti.

La soluzione a questo problema fu individuata disaccoppiando l'accesso al dato, la logica di business e la presentazione dei dati e l'interazione utente.

Il paradigma Model-View-Controller identifica tre componenti fondamentali di una applicazione interattiva:

1. Il Modello, ovvero i dati che si vuole rappresentare e che possono essere modificati dall'utente. Ad esempio, un record di anagrafica rappresenta il Modello di una persona, e può essere mostrato a video, stampato, modificato in vari modi, indipendenti dal contenuto del Modello stesso.

2. Le Viste, ovvero le possibili rappresentazioni del modello, o di parte di esso. Un record di anagrafica può essere rappresentato tramite una form, ma potremmo anche essere interessati a rappresentare solo l'età dei soggetti tramite una barra di un istogramma.

3. Il controllo delle possibilità/modalità di interazione.

Vale la pena di ritornare sul concetto di Controller, perché è indubbiamente il più ostico dei tre, soprattutto per chi è ormai abituato alla standardizzazione imposta dalle interfacce grafiche più diffuse. Quando il paradigma MVC fu introdotto, non vi era alcuno standard riconosciuto che determinasse, ad esempio, il comportamento "normale" di una listbox, ma neppure di una finestra; ad esempio, non vi era un metodo "codificato" per trasferire il focus della tastiera ad una finestra. Alcuni sostenevano la necessità di selezionare la finestra con un click del mouse, altri consideravano sufficiente spostare il cursore del mouse sopra una finestra per renderla attiva; questa situazione perdura tuttora in ambiente X Windows, dove in funzione del window manager selezionato cambia anche la politica di attivazione delle finestre. Di conseguenza, separare il Controllo dell'interazione dalla Vista era non solo naturale, ma anche necessario per conseguire il requisito di *stabilità* dell'applicazione: per cambiare il comportamento di una parte, era sufficiente cambiarne il Controller. Notiamo che ogni parte, per quanto piccola, può avere il suo controller: ad esempio, la listbox aveva il suo controller, così come l'edit box ed ogni altro elemento di interazione.

Il travaso di questo concetti verso il mondo delle architetture web multilivello J2EE fu immediato: *Java Server Pages* per rendere la vista, *Servlet* per realizzare il controller e componenti *EJB* per il modello.

In particolare, uno dei primi pattern nati per filiazione da MVC fu *front controller*, che prevede l'utilizzo di una sola *main servlet* come controller dell'applicazione.

3.2.2 Front Controller

SUN esemplifica questa tecnica nella sua reference application – l'applicazione PetStore – che utilizza una unica *MainServlet*,– il *controller* - col compito di ricevere tutte le GET-POST HTTP per poi delegare l'elaborazione delle stesse a due classi *helper: RequestProcessor* che traduce la richiesta in eventi applicativi e *ScreenFlowManager*, che gestisce le selezioni delle view e della sicurezza.

I dettagli sull'utilizzo di questo pattern si possono trovare in:
http://java.sun.com/blueprints/corej2eepatterns/Patterns/FrontController.html

Un secondo esempio di FrontController è fornito da *Struts*, il framework di presentazione open-source di jakarta dicui si farà cenno più avanti nel documento.

http://jakarta.apache.org/struts/index.html

Nel seguito del documento riportiamo alcuni tra i principali pattern il cui utilizzo è raccomandato nelle specifiche esigenze applicative, concludendo il capitolo con una tabella riepilogativa.

3.2.3 DAO

DAO è un pattern che si utilizza in tutte quelle situazioni in cui può essere opportuno "nascondere" ai client l'implementazione relativa alla sorgente dati utilizzata. Essenzialmente un DAO è un adapter tra il componente e la fointe dei dati. Poiché l'interfaccia esposta dal DAO non cambia se cambia l'implementazione di accesso alla sorgente dati o se cambia la sorgente dati stessa, l'utilizzo di questo pattern consente di cambiare metodo di accesso e/o sorgente dati senza nessua

ripercussione sui componenti che implementano la logica di business. Pertanto il Dao si comporta da adapter tra i componenti applicativi e la sorgente dati

I dettagli sull'utilizzo di questo pattern si possono trovare in:
http://java.sun.com/blueprints/corej2eepatterns/Patterns/DataAccessObject.html

3.2.4 Data transfer object o Value object

Lo scopo di questo pattern è di ridurre l'overhead delle chiamate remote nell'ambito di un'architettura che prevede l'uso di EJB. Infatti se un servizio fornito attraverso l'uso di un EJB deve fornire valori per molte proprietà al presentation layer occorre effettuare tante chiamate remote quanti sono i valori delle proprietà da recuperare. Per ovviare a tutte queste chiamate si incapsulano tutte queste proprietà in un transfer object che viene restituito dall'application layer verso il presentation layer con una sola chiamata remota. Pertanto il presentation layer accede alle varie proprietà attraverso chiamate locali e non remote.

L'uso di questo pattern in un'applicazione che utilizza un framework come 'Struts' consente di disaccopiare gli oggetti di business, quindi l'application layer, dalla presentazione, presentation layer, mantentendo bassi gli impatti derivanti da una eventuale manutenzione evolutiva.

I dettagli di questo pattern si possono trovare in:
http://java.sun.com/blueprints/corej2eepatterns/Patterns/TransferObject.html

3.2.5 Business Delegate

In una applicazione a più layers il presentation layer utilizza dei servizi messi a disposizione dall'application layer. In questo modo i dettagli dei servizi di business sono esposti al presentation layer. Questo crea uno stretto accoppiamento tra i due layers. Pensiamo ad esempio nel caso dell'utilizzo di un enterprise session bean, l'utilizzatore di un tale bean deve essere a conoscenza dei meccanismi di lookup e quindi di naming per avere accesso al bean e quindi al servizio.
Questo pattern nasce quindi dalla necessità di disaccopiare i layers di presentazione e di business e quindi di nascondere i dettagli implementativi del servizio, infatti in tale contesto il business delegate agisce come astrazione locale di un servizio remoto.
Ritornando all'esempio dell'enterprise session bean il componente di presentation che utilizza un servizio remoto diventa completamente trasparente ai servizi di naming e di lookup necessari per accedere ad un'istanza del bean stesso.
Questo pattern usato in congiunzione all'uso del Service locator consente anche un miglioramento delle performance diminuendo l'overhead di chiamate remote per l'utilizzo dei servizi dell'application layer.

I dettagli sull'utilizzo di questo pattern si possono trovare in:
http://java.sun.com/blueprints/corej2eepatterns/Patterns/BusinessDelegate.html

3.2.6 Service Locator

In una applicazione a più layers il presentation layer utilizza dei servizi messi a disposizione dall'application layer, come ad esempio Enterprise JavaBeans (EJB), componenti Java Message Service (JMS). Per interagire con tali componenti il client deve prima localizzarli e poi istanziarli. Ad esempio nel caso di un EJB il client deve prima recuperare l'oggetto che implementa la home interface e poi da tale oggetto recuperare o creare una istanza dell'EJB stesso. Nello stesso modo un client di un JMS deve prima localizzare una Factory di connessioni JMS per ottenere la connessione a una coda JMS. Tutte la applicazioni enterprise utilizzano il servizio di Naming (JNDI) per recuperare e creare EJB, componenti JMS e data source. Ripetuti lookup e creazioni di contesti JNDI sono operazioni che possono causare problemi di performance.
Questo pattern centralizza le operazioni di lookup di servizi distribuiti, pertanto consente di attuare politiche di cache delle risorse con la conseguente riduzione delle operazioni di lookup. Inoltre consente di incapsulare le caratteristiche dipendenti dall'application server del servizio di lookup.

I dettagli sull'utilizzo di questo pattern si possono trovare in:
http://java.sun.com/blueprints/corej2eepatterns/Patterns/ServiceLocator.html

Nella tabella seguente si riassumono i pattern principali con una breve descrizione del contesto di applicazione:

Pattern Name	**Description**	**URL**
Business Delegate [ACM01]	Riduce l'accoppiamento tra gli strati Web ed EJB	http://java.sun.com/blueprints/patterns/BusinessDelegate.html http://java.sun.com/blueprints/corej2eepatterns/Patterns/BusinessDelegate.html
Composite Entity [ACM01]	Modella una rete di entità di business interdipendenti	http://java.sun.com/blueprints/patterns/CompositeEntity.html http://java.sun.com/blueprints/corej2eepatterns/Patterns/CompositeEntity.html
Composite View [ACM01]	Gestisce separatamente il layout ed il contenuto di viste composte multiple	http://java.sun.com/blueprints/patterns/CompositeView.html http://java.sun.com/blueprints/corej2eepatterns/Patterns/CompositeView.html
Data Access Object (DAO) [ACM01]	Astrae ed incapsula i meccanismi di accesso al dato	http://java.sun.com/blueprints/patterns/DAO.html http://java.sun.com/blueprints/corej2eepatterns/Patterns/DataAccessObject.html
Fast Lane Reader	Migliora le performace di lettura di dati tabellari	http://java.sun.com/blueprints/patterns/FastLaneReader.html
Front Controller [ACM01]	Centralizza la processazione delle richieste applicative.	http://java.sun.com/blueprints/patterns/FrontController.html http://java.sun.com/blueprints/corej2eepatterns/Patterns/FrontController.html
Intercepting Filter[ACM01]	Pre e Post-processa le richieste	http://java.sun.com/blueprints/Patterns/InterceptingFilter.html http://java.sun.com/blueprints/corej2eepatterns/Patterns/InterceptingFilter.html

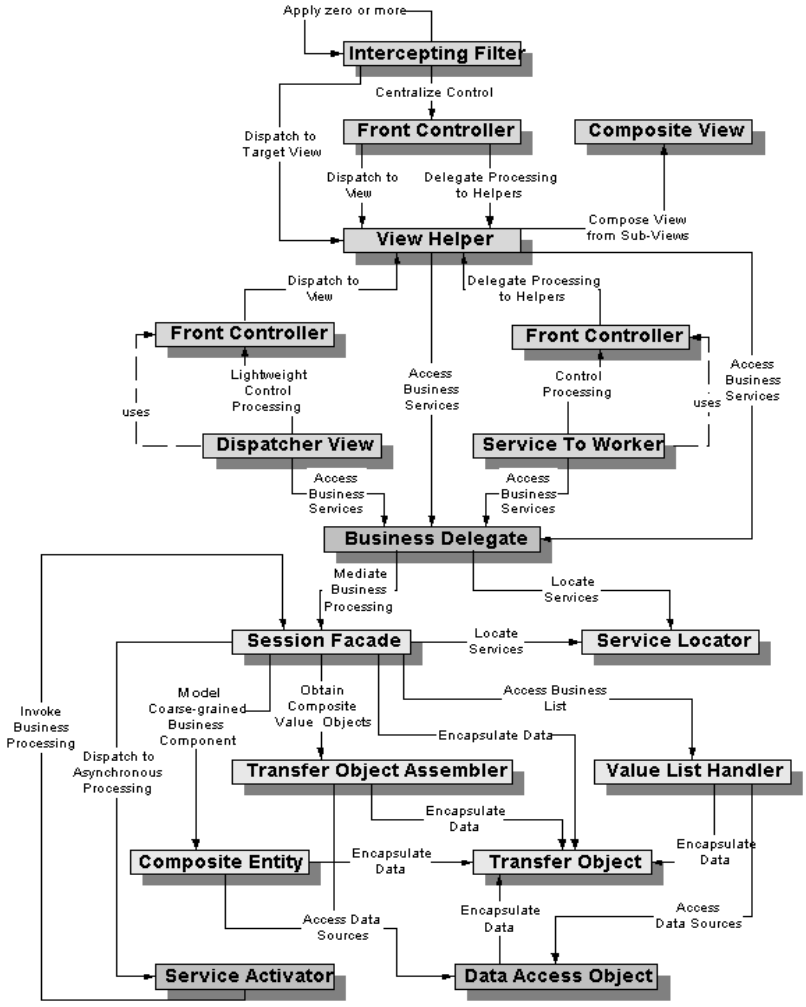
	applicative	
Model-View-Controller	Disaccoppia la rappresentazione dei dati, il comportamento dell'applicazione e la presentazione	http://java.sun.com/blueprints/Patterns/MVC.html
Service Locator [ACM01]	Semplifica l'accesso dei client ai servizi business enterprise	http://java.sun.com/blueprints/Patterns/ServiceLocator.html http://java.sun.com/blueprints/corej2eepatterns/Patterns/ServiceLocator.html
Session Facade [ACM01]	Coordina le operazioni tra molteplici oggetti di business in un workflow	http://java.sun.com/blueprints/Patterns/SessionFacade.html http://java.sun.com/blueprints/corej2eepatterns/Patterns/SessionFacade.html
Transfer Object [ACM01]	Trasferisce dati di business tra gli strati applicativi	http://java.sun.com/blueprints/patterns/TransferObject.html http://java.sun.com/blueprints/corej2eepatterns/Patterns/TransferObject.html
Value List Handler [ACM01]	Itera in maniera efficente una lista virtuale	http://java.sun.com/blueprints/patterns/ValueListHandler.html http://java.sun.com/blueprints/corej2eepatterns/Patterns/ValueListHandler.html
View Helper [ACM01]	Semplifica l'accesso allo stato del modello ed alla logica di accesso al dato	http://java.sun.com/blueprints/patterns/ViewHelper.html http://java.sun.com/blueprints/corej2eepatterns/Patterns/ViewHelper.html

4 Framework

Un *Framework* e' un insieme estendibile di classi e di interfacce, disegnate ed implementate sulla base dei vari design pattern, che collaborano fra di loro al fine di indirizzare la soluzione di un problema tecnico e/o applicativo.

Un framewok non risolve completamente un problema, la risoluzione completa si ha attraverso lo sviluppo di ulteriore codice utente, ovviamente avendo cura di rispettare le regole del framework stesso.

Pur essendo formato da librerie di codice, un framework è molto di più che una semplice libreria di classi. Le differenza tra una libreria e un framework è molto sottile, ma molto importante: una libreria contiene funzioni e routine che possono essere invocate dalle applicazioni, mentre un framework fornisce una serie di componenti che cooperano e che le applicazioni devono estendere per fornire un ben determinato set di funzioni.



Per le applicazioni J2EE presso il MEF si richiede di utilizzare Struts, il framework open source di Apache Software Foundation che implementa il pattern architetturale Model-View-Controller (http://jakarta.apache.org/struts/)

4.1 Navigation Framework - Struts

Come già indicato in precedenza, i concetti MVC (Model, View Controller) si sono tradotti in ambiente Java nell'utilizzo di JSP per *Presentation Layer*. Ma mentre la API J2EE rendono naturale lo sviluppo fintanto che ci si limita al caso delle Web Application, quando si affronta un progetto "enterprise", che debba implementare un framework di navigazione utilizzando le *servlet* come *controller* e EJB come modello si incontrano una serie di problemi la cui soluzione non è banale:

- Mappare i parametri HTTP a dei JavaBean;
- Valicare;
- Gestire errori;
- Internazionalizzare messaggi;
- Evitare gli URI per chiamate JSP hard-coded nel codice.

Struts è uno degli open-source framework che risolve i problemi elencati. Viene di seguito accennata una breve descrizione dei principali componenti del framework.

4.1.1 Servlet Controller

Struts è aderente al pattern MVC in cui la componente primaria del controller è la servlet *org.apache.struts.action.ActionServlet*.

Struts implementa anche il pattern *FrontController* dove la ActionServlet opera come unico punto di controllo che riceve le richieste dal client e, attraverso le definizioni di un file di configurazione (*strut-config.xml*), le mappa su *Actions* applicative.

Le definizioni minime per un mapping sono:

- Request path
- Tipo di oggetto da istanziare

Ogni mapping definisce quindi un path che viene verificato sulla base dell'URI della richiesta ricevuta, ed una classe (classe Java che deve estendere Action) responsabile per gestire la business logic associata alla richesta.

4.1.2 Action Objects

Gli oggetti *Action* gestiscono le richieste e rispondono al client, pertanto elaborano la richiesta del client e possano il controllo (*forward*) o d un'altra *Action* oppure a una vista che può essere una pagina statica html, oppure una java server page.
Gli oggetti *Action* sono associati al controller e quindi hanno accesso ai metodi della servlet. Nel passare il controllo, un oggetto può indirettamente passare uno o più oggetti condivisi, inclusi JavaBeans e FormBeans, ponendoli in uno dei contesti condivisi.
Gli oggetti Action devono derivare da **org.apache.struts.action.Action** e riscrivere il motodo **execute**.

Nelle applicazioni *Struts* la business logic viene implementata all'interno del metodo execute della classe Action.

4.1.3 Form Beans

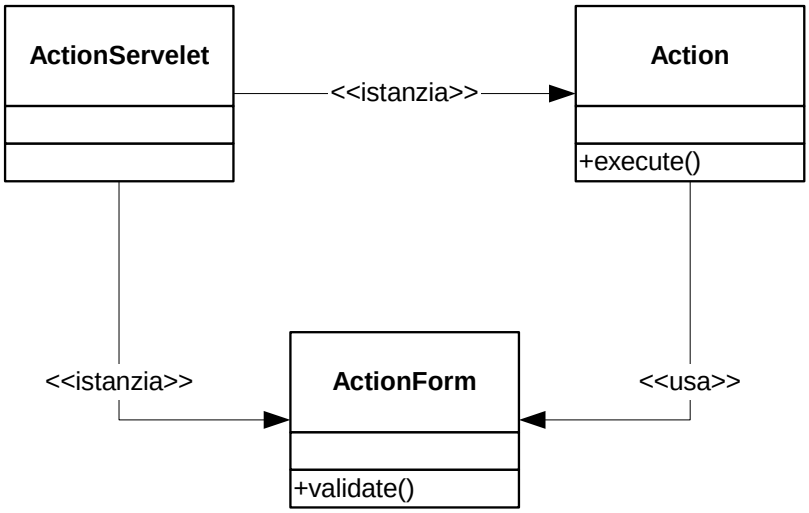
I JavaBeans possono anche essere usati per gestire FORM di input. Un problema chiave nel disegnare delle Web applications è il mantenimento e la validazione dei dati immessi dall'utente tra una richiesta e l'altra.

Con *Struts* si può facilmente mappare i dati di un FORM HTML di input in un *FormBean.* Ogni bean predisposto per la gestione e validazione di una form HTML deve derivare dalla classe **org.apache.struts.action.ActionForm.** Il bean può essere salvato in uno dei contesti condivisi (page, request, session) così da poter essere usato da altri oggetti. Il *FormBean* può essere quindi usato:

- Per recuperare i dati inseriti nella FORM HTML dall'utente
- Per effettuare la validazione dei dati inseriti dall'utente
- Dalla JSP per ri-popolare i campi di un FORM HTML

In caso di errori di validazione *Struts* ha un meccanismo condiviso per lanciare e visualizzare messaggi di errore. In tale caso i controlli di validazione sui campi devono essere effettuati riscrivendo il motodo validate della classe ActionForm.
Struts invoca automaticamente (se impostato nel file di configurazione) tale metodo di validazione ogni volta che la JSP che contiene un FORM associato ad un *ActionForm* fa il *Submit* della richiesta. Può essere fatta qualunque tipo di validazione, l'unico requirement è che venga ritornato un oggetto di tipo *ActionErrors* che è una collection di oggetti *ActionError*. Ogni *ActionError* corrisponde ad un singolo errore di validazione che viene mappato, attraverso un file esterno, al messaggio di errore da visualizzare.

Un *FormBean* è definito nel file di configurazione (struts-config.xml) ed associato ad una Action. Quando una richiesta invoca una certa *Action* che usa un FormBean, la *controller* servlet recupera da uno dei contesti condivisi o crea il FormBean e lo passa all'oggetto *Action.*



4.1.4 Custom tags

Struts include quattro *JSP Tag Libraries*:

- ***HTML Tag Library***, il quale include tags per descrivere pagine dinamiche, specialmente forms.
- ***Beans Tag Library***, che fornisce tags addizionali per la gestione dei JavaBeans e per la gestione dell'Internazionalizzazione.
- ***Logic Tag Library***, per il supporto di esecuzione condizionata e loops.
- ***Template Tag Library***, per produrre ed usare template JSP comuni in pagine diverse

5 Log Applicativo e di Servizio

Ogni applicazione ha la necessità di effettuare attività di logging, sia in fase di sviluppo a scopo di debugging, sia in produzione quando una corretta gestione dei log può aiutare gli amministratori del sistema ad individuare le cause di possibili anomalie.

Per evitare che ogni gruppo di lavoro inventi un proprio tool di logging si dovrebbe scegliere un framework di logging a cui tutti i gruppi di lavoro dovrebbero attenersi.
Si propone qui di seguito il tool open source LOG4J, ma si è liberi di utilizzare un qualsiasi tool di logging purchè il risultato sia equivalente a quello che qui di seguito verrà riportato.
In particolare, tutte le applicazioni dovranno avere la possibilità di attivare, in caso di necessità, funzionalità di "logging" paramentrizzabili e dotate di vari livelli di "auditing", altrimenti "dormienti".

5.1 Log4j

Tools: Log4j http://jakarta.apache.org/log4j/docs/index.html

Attualmente non esistono API standard anche se SUN, con la nuova release 1.4, introdurrà le *Java 1.4 Logging API* basato su *JSR47* di Graham Hamilton. Esiste però un progetto open-source che va sotto il nome di *Log4J* che offre ottime funzionalità per implementare una sofisticata strategia di logging.
Una delle caratteristiche principali di Log4J è il grado di configurablità ed estendibilità del pacchetto.
Infatti una volta che gli statement di log sono stati inseriti nel codice possono essere controllati attraverso il file di configurazione. Pertanto a runtime il logging può essere abilitato o disabilitato, può essere modificato il canale di output e così via. In sostanza log4j è disegnato in maniera tale che le istruzioni di log restino nel codice di produzione senza che ci siano scadimenti di performance. Questo perché utilizza un *PropertyConfigurator* per la propia configurazione inizializzato a partire da un file di properties del tipo:

Esempio di file di configurazione:

```
#Valorizzo il logger radice al livello di debug e definisco due appenders stdout e R
log4j.rootLogger=ALL, stdout, R, html

#Specifico le caratteristiche dei due appender
#stdout di tipo console appender
log4j.appender.stdout=org.apache.log4j.ConsoleAppender

#definisco la classe del pattern layout
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout

#definisco il pattern del log
log4j.appender.stdout.layout.ConversionPattern=%p [%t] (%F:%L) - %m%n

#R di tipo file ricorsivo
log4j.appender.R=org.apache.log4j.RollingFileAppender

#nome del file di log
log4j.appender.R.File=esempio.log

#definisco la massima dimensione del file di log
log4j.appender.R.MaxFileSize=100KB
```

```
#definisco il numero di copie di backup da conservare
log4j.appender.R.MaxBackupIndex=1

#definisco la classe del pattern layout
log4j.appender.R.layout=org.apache.log4j.PatternLayout

#definisco il pattern del log
log4j.appender.R.layout.ConversionPattern=%p %t %c - %m%n
# html di tipo file
log4j.appender.html=org.apache.log4j.FileAppender
#definisco il nome del file
log4j.appender.html.File=out.html

#definisco il layout utlizzando la classe HTMLLayout
log4j.appender.html.layout=org.apache.log4j.HTMLLayout
#definisco se voglio o  meno le inforamzioni del file java e della linea di codice che hanno emesso
#l'evento di logging
log4j.appender.html.layout.LocationInfo=true
#definisco il titolo del file html
log4j.appender.html.layout.Title=Log4j utilizzando HTMLLayout
```

Log4J organizza i componenti (*categories)* in modo gerarchico. La gerarchia viene identificata attraverso i vari livelli di packaging del codice fino a poter identificare, a livello più basso, la singola classe.
Per ogni categoria si può definire la priorità del logging (FATAL, ERROR, DEBUG, WARN, INFO) ed una destinazione del messaggi di log implementate da delle classi Java dette *Appenders*.

Log4J fornisce una serie di appenders tra i quali:

- Text file
- Rolling log files
- UNIX system logs
- SMTP Appender

Oltre alla destinazione viene, infine, definito il formato degli output attraverso classi di formattazione.
Esistono diverse classi di layout:

- SimpleLayout
- XMLLayout
- PatternLayout
- HTMLLayout

Il PatternLayout consente di formattare l'output dell'evento di logging scegliendo sia quali informazioni riportare sia il loro formato di output.

Vediamo in dettaglio due possibili esempi di pattern presenti nel file di configurazione:

```
log4j.appender.stdout.layout.ConversionPattern=%p [%t] (%F:%L) - %m%n
```

Vediamo in dettaglio il significato dei vari simboli che costituiscono il pattern:

- %p stampa la priorità dell'evento di logging
- %t stampa la thread che ha generato l'evento di logging
- %F stampa il nome del file java che ha generato l'evento

- %L stampa il numero di riga all'interno del file java che ha generato l'evento

Ecco il risultato di un tale conversion pattern:

INFO [main] (ProvaLog4J.java:28) – Avvio dell'applicazione.

```
log4j.appender.R.layout.ConversionPattern=%p %t %C - %m%n
```

Vediamo in dettaglio il significato dei vari simboli che costituiscono il pattern:

- %p stampa la priorità dell'evento di logging
- %t stampa la thread che ha generato l'evento di logging
- %c stampa il nome completo comprensivo del package della classe che ha emesso l'evento
- %m stampa il messaggio associato all'evento
- %n inserisce un carattere di new line dipendente dal sistema

Ecco il risultato di un tale conversion pattern:

INFO main it.mef.ProvaLog4J – Avvio dell'applicazione.

E' possibile addirittura definire dei Renderer che derivano da *ObjectRenderer* che producono un output in funzione della classe di cui si richiede il log.

Esempio d'uso:

```
package it.mef;

import org.apache.log4j.Logger;
import org.apache.log4j.PropertyConfigurator;

public class ProvaLog4J
{
  //istanzio un logger associandogli il nome della classe che lo istanzia
  static Logger logger = Logger.getLogger(ProvaLog4J.class.getName());

  public static void main(String[] args)
  {
    //inizializzo il logger con il file di properties passato come argomento
    PropertyConfigurator.configure(args[0]);

    //emetto due eventi logging
    logger.info("Avvio dell'applicazione.");
    logger.info("Fine dell'applicazione.");
  }
}
```

Output in console:

```
INFO [main] (ProvaLog4J.java:26) - Avvio dell'applicazione.
INFO [main] (ProvaLog4J.java:27) - Fine dell'applicazione.
```

Output sul file esempio.log:

```
INFO main it.mef.ProvaLog4J - Avvio dell'applicazione.
INFO main it.mef.ProvaLog4J - Fine dell'applicazione.
```

Output sul file esempio.html:

Log session start time Sun Dec 21 18:57:26 CET 2003

Time	Thread	Level	Category	File:Line	Message
0	main	INFO	it.mef.ProvaLog4J	ProvaLog4J.java:26	Avvio dell'applicazione.
10	main	INFO	it.mef.ProvaLog4J	ProvaLog4J.java:27	Fine dell'applicazione.

6 Web Services

Fare riferimento all'attuale implementazione Consip delle specifiche AIPA relativa alla porta delegata e alla porta di dominio.

7 Servizi Applicativi

In questo capitolo saranno fornite alcune indicazioni relative ad aspetti, comuni a tutte le applicazioni Web realizzate per il MEF, per i quali è necessario un approccio uniforme, in modo da facilitare e rendere più efficace le attività di manutenzione e di sviluppi di nuove funzionalità.

7.1 Proprieta' Infrastrutturali e Applicative

Ogni applicazione sviluppata ha sempre a che fare con la necessità di non cablare all'interno del codice parametri/valori che possono o devono essere modificati col passare del tempo o devono essere modificati in base all'ambiente dove il software deve operare (test o produzione).
Il ricorso a risorse esterne (file o DB) su cui memorizzare tali valori è quindi d'obbligo.

7.1.1 Uso di files di properties

Un approccio molto semplice è quello di utilizzare un file di properties esterno che viene letto all'inizializzazione dell'ambiente e caricato e mantenuto in memoria da una classe che implementi il Singleton pattern oppure una classe con attributi statici per la memorizzazione dei valori di properties.
Supponiamo per esempio uno scenario in cui all'inizializzazione di una Servlet di startup viene caricato un file di properties in una classe con attributi statici.

Esempio di classe statica per mantenere e fornire ai clients le informazioni del file di properties:

```
import java.util.Properties;

public class Proprieta
{
  //attributo statico di tipo Properties per la memorizzazione come chiave, valore
  private static Properties pParametriIniz;

  //metodo statico che ha come parametro una stringa che rappresenta una key e che restituisce
  //il value associato a quella key
  public static String getParametro(String pNomePar)
  {
    return pParametriIniz.getProperty(pNomePar);
  }

  //metodo statico che ha come parametro una collezione di coppie key, value e che memorizza in un
  //campo statico tali coppie
  public static void setProperties(Properties pParametri)
  {
    pParametriIniz = pParametri;
  }
}
```

Vediamo ora una possibile implementazione del metodo di inizializzazione della servlet, in questo caso si suppone che il path realtivo alla root della web application sia salvato in un parametro di contesto all'interno del file di configurazione ***web.xml***.

Esempio di file web.xml:

```
<web-app>
 <display-name>NomeApplicazione</display-name>
 <context-param>
  <param-name>file_properties</param-name>
  <param-value>WEB-INF/properties/<nomeApplicazione>.properties</param-value>
</context-param>
…
```

Esempio metodo init della servlet:

```
import java.io.FileInputStream;
import java.util.Properties;
import javax.servlet.ServletContext;
import javax.servlet.ServletException;

public class ServletInizializzazione extends HttpServlet
{
  public void init(ServletConfig config) throws ServletException
  {
    //ricavo il servlet context dove il servlet container ha memorizzato il path relativo del file
    //di properties come indicato nel file web.xml
    ServletContext pCont = config.getServletContext();
    //ricavo il valore del path relativo
    String pPath = pCont.getInitParameter("file_properties");

    //determino il path reale rispetto al file system per aprire il file
    String pRealPath = pCont.getRealPath(pPath);

    //istanzio un oggetto di tipo Properties dove verranno caricate le coppie key, value
    Properties pParametri = new Properties();

    try
    {
      //apro il file fisico
      FileInputStream pFileStream = new FileInputStream(pRealPath);
      //carico il file
      pParametri.load(pFileStream);
      //chiudo il file
      pFileStream.close();
    }
    catch (Exception e)
    {
      logger.error("Errore nel caricamento dei parametri di inizializzazione " + e.getMessage());
    }

    //chiamo il metodo statico sulla classe Proprieta per memorizzare le coppie key, value
    //appena caricate
    Proprieta.setProperties(pParametri);
  }
}
```

Esempio di file di properties:

```
...
dataSourceName=java:jdbc/DataSource
dateFormatJava=dd-MMM-yy HH:mm:ss
...
```

Inizializzazione in ambiente Struts

Nel caso di una applicazione Struts il supporto per questo tipo di inizializzazioni è demandato all'uso di Plugins. Facendo riferimento alla stessa classe statica per il caricamento delle properties, allo stesso file di configurazione **web.xml** dell'esempio precedente vediamo un esempio di Plugin.

Esempio di file di configurazione struts-config.xml:

```
<struts-config>
…
  <plug-in className="PluginInizializzazione"/>

</struts-config>
```

Esempio di Plugin struts:

```
import java.io.FileInputStream;
import java.util.Properties;
import javax.servlet.ServletContext;
import javax.servlet.ServletException;

import org.apache.struts.action.ActionServlet;
import org.apache.struts.action.PlugIn;
import org.apache.struts.config.ApplicationConfig;

public class StartupPlugin implements PlugIn
{
  public StartupPlugin()
  {
    super();
  }

  public void destroy()
  {
  }

  public void init(ActionServlet pServlet, ApplicationConfig pAppConfig) throws ServletException
  {
    ServletContext pCont = pServlet.getServletConfig().getServletContext();
    String pPath = pCont.getInitParameter("file_properties");
    String pRealPath = pCont.getRealPath(pPath);

    Properties pParametri = new Properties();
    try
    {
      FileInputStream pFileStream = new FileInputStream(pRealPath);
      pParametri.load(pFileStream);
      pFileStream.close();
    }
    catch (Exception e)
    {
      logger.error("Errore nel caricamento dei parametri di inizializzazione " + e.getMessage());
    }

    Proprieta.setProperties(pParametri);
  }
}
```

7.1.2 Standards dei file di inizializzazione di properties

I files di inizializzazione dovranno seguire alcune convenzioni sia per quanto concerne la loro locazione fisica sia per ciò che riguarda la loro denominazione. Per quanto riguarda il primo aspetto tutti i files di properties dovranno trovarsi in una cartella denominata **properties** all'interno della cartella standard **WEB-INF**, mentre il loro nome dovrà essere composto nel seguente modo: <nomeApplicazione><nomeIdentificativoFile>.properties.

7.2 Exceptions

La gestione delle eccezioni applicative dovrebbe essere realizzata a partire da una exception di base da cui vengono derivate tutte le exceptions specifiche. In questo modo possono essere filtrati gli errori di tipo applicativo da quelli di sistema in ogni blocco di codice.

8 Organizzazione Packages e File System

Come descritto nei capitoli precedenti si sono elencati una serie di modelli di disegno che suddividono in modo sufficientemente preciso le responsabilità tra diversi strati applicativi (layers) e le modalità con cui questi interagiscono tra loro.
Questa suddivisione logica dovrebbe rispecchiarsi, in fase di sviluppo, in una organizzazione dei packages e delle classi.
Inoltre lo standard J2EE introduce una serie di specifiche che "obbliga" lo sviluppatore ad adottare alcune convenzioni.

8.1 Naming convention per i packages

Le seguenti convenzioni di nomenclatura vanno attuate al fine di standardizzare riferimenti a metodi, classi, variabili, etc…, per permettere, attraverso lo studio di un semplice identificativo, di risalirne alla tipologia e, possibilmente, alla funzione, anche in presenza di metodi eccessivamente prolissi, ove la sezione iniziale concernente, ad esempio, le dichiarazioni di variabili con scope interno, si trovi su una pagina precedente.

La seguente tabella riporta, per ogni tipologia di identificativo, le relative convenzioni di nomenclatura, assieme ad esempi, se necessario.

Identificativo	Nomenclatura	Esempio
packages	Il prefisso di un identificativo di package va scritto in caratteri ASCII minuscoli.	`sun.com.eng`

classi e interfacce	Gli identificati di classe o interfaccia devono essere formati da uno o più sostantivi, con combinazioni maiuscolo/minuscolo relative alle iniziali dei sostantivi che lo compongono. Tali identificativi devono risultare più semplici e descrittivi possibile. Utilizzare parole intere, evitare acronimi ed abbreviazioni laddove non siano implicitamente ovvie (URL, HTML, etc…), in nessun caso inserire caratteri non alfanumerici.
	
	Il primo carattere di un identificativo di classe o interfaccia deve essere sempre maiuscolo.
		`class Raster;`
		`class ImageSprite;`
		`interface Storing;`
		`interface TestURL;`
		`interface StampaTest;`

metodi	Gli identificativi di metodo devono essere	`run();`

	formati da uno o più verbi, scelti fra i più descrittivi in relazione alla funzionalità del metodo stesso, con combinazioni maiuscolo/minuscolo soggette alle stesse regole valide per gli identificativi di classe o interfaccia. Anche negli identificativi di metodo non è ammissibile l'inclusione di caratteri non alfanumerici.
	
	Il primo carattere di un identificativo di metodo deve essere sempre minuscolo.
		`runFast();`
		`stampaPagina();`
		`eseguiTest();`

variabili	Gli identificativi di variabile seguono le stesse regole degli identificativi di metodo, per qualsiasi tipologia di dato siano esse riferite, ad eccezione delle costanti. Identificativi di variabile formati da una sola lettera devono essere evitati, a meno che non indichino variabili di uso temporaneo limitato nello scope.	`int a;`
		`int b;`
		`float ilMioFloat;`

costanti	Gli identificativi di costante consistono in caratteri ASCII esclusivamente maiuscoli, con parole od acronimi separati dal carattere di sottolineatura.	`(static final int)`
		`MAX_LEN = 3;`
		`MIN_LEN = 0;`

La denominazione dei packages adotterà le seguenti regole:

Per i package di uso generale cross application:
it.mef.<nome package>.

Per le singole applicazioni si deve seguire la seguente naming convention:

it.mef.<nome applicazione>.

Nel caso si utilizzasse il pattern Dao, tutti gli oggetti che contengono query sql dovranno trovarsi in un package con il seguente nome:

it.mef.<nome applicazione>.dao

Nel caso si utilizzasse il pattern Value object, tutti gli oggetti che rappresentano value object dovranno trovarsi in un package con il seguente nome:

it.mef.<nome applicazione>.valueobject o **it.mef.<nome applicazione>.transferobject**

Nel caso si utilizzasse il pattern Business delegate, tutti gli oggetti che rappresentano value object dovranno trovarsi in un package con il seguente nome:

it.mef.<nome applicazione>.businessdelegate

E' necessario poi nella definizione dei packages evitare le *referenze circolari*. Se due classi si referenziano l'un l'altra, il risultato è una dipendenza circolare: nessuna delle due classi può funzionare senza l'altra e ne consegue che nessuna delle due e riusabile separatamente. In generale un ridisegno può evitare queste dipendenze. Nel caso in cui questa mutua referenza sia necessaria è conveniente che le due classi condividano lo stesso package.

8.2 Struttura Directory J2EE

La fase di sviluppo di una Web Application J2EE viene fatta su una struttura di directory ben definita così da poter essere successivamente archiviata e distribuita su qualunque server compatibile J2EE
Le specifiche J2EE forniscono istruzioni per la costruzione, il packaging e successiva distribuzione (deploying) di una applicazione J2EE.

Una applicazione J2EE consiste quindi di un insieme di moduli. I moduli vengono archiviati nel formato JAR indipendentemente dal tipo di modulo.

L'estensione che viene data all'archivio identifica il tipo di modulo:

- Moduli EJB sono archiviati in JAR files.
- Moduli Web sono archiviati in Web Archivi (WAR) files.
- Moduli Application Client sono archiviati in JAR files.
- Enterprise applications sono archiviati in Enterprise Archivi (EAR) files.

Oltre ai moduli esistono poi i cosiddetti *Deployment Descriptor*s contenenti le informazioni relative ai moduli. Alcuni di questi sono standard e saranno gli stessi per tutti gli application server J2EE. Altri sono proprietari e includono definizioni non incluse in specifiche J2EE che dipendono dal tipo di Application Server che viene utilizzato.

Esempio EAR

File	Descrizione
/META-INF/application.xml	Il deployement descriptor dell'applicazione
/META-INF/MANIFEST.MF	Standard JAR file manifest
/client_module.jar	
/ejb_module.jar	Modulo Ejb contentente gli enterprise javabean utilizzati dall'applicazione (Application layer)
/helper_classes.jar	File jar contentente le helper classes utilizzate nell'applicazione
/web_module.war	Il modulo web contenente i componenti web dell'applicazione (Presentation layer)

Per quanto riguarda la struttura di una enterprise application si deve far riferimento alle specifiche Sun 'Java™ 2 Platform Enterprise Edition Specification v1.4' e in particolare a quanto indicato nel capitolo 'J2EE.8'.
Tale documento di specifiche è reperibile su sito Sun al seguente indirizzo:
http://java.sun.com/j2ee/download.html

Esempio WAR

Tutte le servlets, le classi, i file statici, e le altre risorse appartenenti ad una Web Application sono organizzati in una struttura gerarchica di directory. La *root* di questa gerarchia definisce la *document root* della Web Application. Tutti i file al di sotto della *root* directory possono essere utilizzati dal client eccetto quelli all'interno della directory WEB-INF, anch'essa sotto la *root* directory.

File	Descrizione
/web_module.war/	*Document root*
/web_module.war/META-INF/MANIFEST.MF	Standard JAR file manifest
/web_module.war/WEB-INF/web.xml	Il deployment descriptor del modulo web
/web_module.war/WEB-INF/classes/MyServlet.class	Una classe che rappresenta una servlet
/web_module.war/WEB-INF/lib/*.jar. *.properties	Librerie java utilizzate a runtime
/web_module.war/*.jsp, *.html, *.gif, *.js, *.css	Risorse statiche e JSPs

Per quanto riguarda la struttura di una web application si deve far riferimento alle specifiche Sun 'JSR-000154 Java™ Servlet 2.4 Specification' e in particolare a quanto indicato nel capitolo 'SRV.9'.
Tale documento di specifiche è reperibile su sito Sun al seguente indirizzo:
http://java.sun.com/products/servlet/download.html

E' importante sottolineare che se i files di deploy sia di tipo **.war** che **.ear** non rispondono allo standard J2EE non sarà possibile effettuare il deploy nell'application server.

9 Linee Guida Sviluppo

In questo capitolo verranno fornite alcune linee guida per lo sviluppo di codice java, pagine web, e risorse associate ad applicazioni web in genere (per esempio HTTPSession, DataSource, ecc.).

Ovviamente, non si intende essere in alcun modo esaustivi: le guidelines vogliono sono un punto di partenza ed assieme uno stimolo a sviluppare particolare sensibilità per aspetti di performance.

9.1 Struttura del codice sorgente

Viene qui descritta la struttura generica di un file sorgente Java, con i blocchi funzionali espressi nell'ordine in cui devono apparire all'interno del listato stesso.

9.1.1 Commenti Iniziali

Ogni file sorgente deve iniziare con un commento c-style ove figurino il nome della classe, le informazioni relative al versioning del sorgente, l'autore/i e la data di codifica:

```
/*
 * nome classe
 *
 * informazioni di versioning
 *
 * data di codifica
 *
 * autore/i
 */
```

9.1.2 Dichiarazioni di Classi o Interfacce

La dichiarazione di una classe o interfaccia deve essere obbligatoriamente composta dai seguenti campi, nell'ordine in cui vengono qui riportati:

- commento di documentazione di classe o interfaccia

  commento per la generazione di documentazione di classe o interfaccia, v. par. "Commenti", sez. "Commenti di Documentazione".

- statement `class` o `interface`
- commento di implementazione di classe o interfaccia

  commento recante informazioni valide per l'intera classe o interfaccia, non appropriate per l'inclusione nel commento di documentazione iniziale espresso al primo punto.

- variabili statiche (`static`)

  l'ordine di dichiarazione è il seguente:

  `public`

  `protected`

  `package-level (nessun access modifier)`

  `private`

- variabili di istanza

  dichiarate con lo stesso ordine delle variabili statiche (v. punto precedente).

- costruttori
- metodi

  l' ordine di raggruppamento dei metodi è esclusivamente funzionale e non tiene conto del relativo scope. Questo allo scopo di favorire la leggibilità del codice, mantenendo evidente il flusso operativo.

9.2 Indentazione

L'unità standard di indentazione utilizzata è di quattro spazi. L'esatta costituzione dell'indentazione (caratteri di spaziatura o di tabulatura) rimane a discrezione dello sviluppatore. Il carattere di tabulatura deve essere impostato a otto spazi.

9.2.1 Lunghezza di Linea

Evitare una lunghezza di linea superiore ad 80 caratteri, cercando di attenersi, ove possibile, ad un limite massimo di 70 caratteri.

9.2.2 Interruzione di Linea

Quando un'espressione non riesce ad essere compresa su di un'unica linea di codice, si rende necessario interromperla seguendo le seguenti indicazioni:

- interrompere dietro una virgola o prima di un operatore, allineando la nuova linea all'inizio dell'espressione della linea precedente

```
metodo(parametro1,parametro2,
        parametro3);
```

- sono preferibili interruzioni di alto livello piuttosto che basso. Nel seguente esempio la prima forma di interruzione è preferita poiché avviene all'esterno dell'espressione racchiusa tra parentesi, che gode di un livello superiore.

```
var1 = var2 * (var3 + var4 – var5)
       + 4 * var6;

var1 = var2 * (var3 + var4
       – var5) + 4 * var6;
```

- se le regole sopra espresse portano alla generazione di codice non facilmente leggibile, procedere con una semplice indentazione di 8 spazi

```
//indentazione convenzionale
metodo(int arg1, int arg2, int arg3,
       int arg4) {
    ...
}

//evitare livelli di indentazione eccessivamente profondi con 8 spazi
private static synchronized nomediunmetodomoltolungo(Object parametro1,
        Object parametro2, Object parametro3, Object parametro4) {
            ...
}
```

Nota:

Eccezioni alle regole appena espresse sussistono nel caso di interruzione di linea per statements `if` ed operatori ternari. Mentre i blocchi `if` utilizzano una indentazione di 8 spazi, poiché il livello convenzionale di indentazione ne renderebbe il codice di difficile lettura, per gli operatori ternari non sussiste la regola di interruzione di linea prima di un operatore, ma viene anzi applicato il contrario.

Non utilizzare questa indentazione in caso di interruzione di linea, poiché la chiamata al metodo potrebbe non essere notata leggendo il codice:

```
if ((condizione1 && condizione2)
           || (condizione3 && condizione4)
           || (condizione5 && condizione6)) {
    metodo();
}
```

La corretta indentazione in caso di interruzione di linea è invece la seguente:

```
if ((condizione1 && condizione2)
             || (condizione3 && condizione4)
             || (condizione5 && condizione6)) {
         metodo();
}
```

In caso di interruzione di linea all'interno di operatori ternari, procedere in uno dei seguenti modi:

```
var = (nomeditipobooleanomoltolungo) ? valore1
                                     : valore2;

var = (nomeditipobooleanomoltolungo)
      ? valore1
```

9.3 Attenzioni generiche

- evitare ogni istanza di classe o variabile pubblica non indispensabile.
- evitare di istanziare classi per referenziare variabili e/o metodi statici.
- evitare assegnamenti di variabile multipli sulla stessa riga, come nel seguente esempio:

```
int variabile1 = variabile2 = variabile3 = 4;
```

- evitare assegnamenti compatti come nel seguente esempio:

```
d = (a = b + c) + r;
```

tale riga deve essere scritta come segue:

```
a = b + c;
d = a + r;
```

- utilizzare i raggruppamenti con parentesi tonde quando possibile, al fine di eliminare il più possibile dubbi su precedenze fra operatori diversi.

  evitare la seguente riga:

```
if (a == b && c == d)
```

  scriverla invece come segue:

```
if ((a == b) && (c == d))
```

- utilizzare operatori ternari, ove possibile.

  evitare la seguente pratica di codifica:

```
if (condizione) {
    return x;
```

```
}
return y;
```

scriverla invece come segue:

```
return (condizione ? x : y);
```

- le espressioni contenenti operatori binari, presenti prima del carattere '?' in un operatore ternario, devono essere racchiusi fra parentesi tonde, come nel seguente esempio:

```
(x >= 0) ? x : -x;
```

la ripetizione di righe consecutive di codice all'interno dello stesso package è inammissibile. Tali casi vanno affrontati parametrizzando idoneamente un metodo atto a fornire la funzionalità richiesta. Non sono accettabili porzioni di codice ripetute in cui i cambiamenti minimi sarebbero potuti essere gestiti attraverso una corretta parametrizzazione del metodo.

- non è accettabile l'utilizzo di classi o packages di terze parti dei quali non si sia in possesso del codice sorgente, e in casi in cui detto codice non sia allineato con le convenzioni standard di sviluppo oggetto di questo documento. In particolare non è ammissibile l'inclusione di packages proprietari di terze parti quale effetto collaterale derivato dall'utilizzo di particolari strumenti di sviluppo.
- la spaziatura del codice non è soggetta a regole arbitrarie, ma deve seguire le linee guida qui espresse. In particolare non è accettabile inserire linee vuote a piacimento all'interno del codice, a meno che non rispettino quanto espresso nel par. "Spaziatura", sez. "Linee Vuote".
- l'inclusione di codice commentato all'interno di un file sorgente non è accettabile. Il codice non funzionante, o isolato per altri motivi, va incondizionatamente rimosso.
- evitare di utilizzare la lingua inglese all'interno del codice. Ogni riferimento a variabili, istanze, o identificativi in genere, deve essere scritto utilizzando la lingua italiana. Tale regola vale ovviamente anche per commenti, siano essi di implementazione, siano (a maggior ragione) di documentazione. L'estrazione di documentazione di codice in lingua inglese, tramite il tool javadoc, non è accettabile.

9.4 Java Performance

- **Concatenazione String '+=':**
  L'uso di questo operatore di concatenazione impatta fortemente le performance. Il metodo migliore per la concatenazione di stringhe consiste nell'utilizzo di java.lang.StringBuffer con il metodo append(String) per concatenare e il metodo toString() per ottenere la stringa risultato della concatenazione
- **Interfaccia Serializable (Tag interface):**
  Per avere la possibilita' di attivare la persistenza delle sessioni con WebSphere e' necessario che ogni oggetto memorizzato nella HttpSession implementi l'interfaccia java.lang.Serializable. Inoltre tutti gli oggetti che devono transitare tra Presentation layer e Application layer devono implementare java.lang.Serializable per consentire l'eventuale transito degli oggetti stessi tra VM differenti che potrebbero risiedere su dispositivi fisici differenti
- **Minimizzare l'uso di System.out.println():**
  Strumento utile in fase di debugging e unit testing, ma con costi alti in termini di uso risorse. Si consiglia di utilizzare framework di logging idonei allo scopo e che possono essere attivati o disattivati tramite file di properties esterni all'applicazione, vedi ad esempio Log4J

- **Minimizzare l'uso di java.util.Hashtable:** Hashtable e Properties hanno i metodi di accesso sincronizzato. I metodi sincronizzati rappresentano punti di serializzazione delle chiamate questo per garantire l'assenza di corse critiche all'interno del metodo. Si consiglia, nel caso di accesso concorrente in lettura, di considerare l'uso di altre classi, quali java.util.HashMap
- **Minimizzare l'uso di new:** La creazione di oggetti in Java e' un operazione costosa in un'applicazione con un numero elevato di threads. Ove possibile, si consiglia di riusare oggetti, anche attraverso l'implementazione di metodi che ne ripristino lo stato iniziale

9.5 HTTPSession

- **Non memorizzare "grandi" oggetti nella HttpSession:**
  Applicazioni che richiedono la persistenza della sessione (per esempio per garantire meccanismi di fail-over e ripartenza) necessitano di effettuare la persistenza su DB di HttpSession. Questo comporta la serializzazione e la deserializzazione degli oggetti presenti in sessione (con ovvi costi di performance). Valore consigliato per performance ottimali e' 4 KBytes. Per dare un indicazione di massima, sessioni di 16KB rispetto a sessioni di 4 KB hanno un costo medio di performance del 100% superiori
- **Rilascio HttpSession e/o rimozione degli oggetti memorizzati:**
  Rilasciare applicativamente la sessione appena possibile (per esempio in fase di uscita dall'applicazione) utilizzando javax.servlet.http.HttpSession.invalidate(), in modo da non appesantire l'Application Server nella serializzazione di HttpSession in memoria e/o su dDB. Nel caso non si possa invalidare l'intera sessione è possibile effettuare l'unbouinding di singoli oggetti dalla sessione attraverso l'utilizzo del metodo javax.servlet.http.HttpSession.removeAttribute(java.lang.String name)
- **Non creare HttpSession per default in JSPs:**
  Esplicitare la non creazione, quando non necessaria, all'interno di JSPs tramite la direttiva: <%@ page session="false"%>

9.6 DataSource

- **Utilizzo dei Datasource:**
  L'acquisizione di connessioni verso i database deve essere effettuata, esclusivamente, attraverso l'utilizzo dei datasources definiti a livello di application server e resi disponibili attraverso la lookup sul servizio di naming (JNDI). Il datasource implementa politiche di pooling delle connessioni che posso essere variate in modo dichiarativo e quindi presentano caratteristiche di scalabilità al crescere delle necessità applicative senza dover modificare il codice applicativo
- **Riusare Datasource:**
  Evitare l'acquisizione di datasource per ogni accesso al database, in quanto l'operazione di lookup sul servizio di naming (JNDI) risulta costosa in termini di performance e scalabilita'. Pertanto è buona norma acquisire il datasource una sola volta o in fase di inizializzazione dell'applicazione o al primo accesso al database
- **Connection pooling:**
  E' espressamente sconsigliato l'utilizzo di connection pooling applicativi o l'acquisizione di connessioni attraverso il caricamento di JDBC driver e successiva invocazione del metodo java.sql.DriverManager.getConnection(…)

- **Rilascio delle risorse:**
  Chiudere esplicitamente le connessioni dopo il loro utilizzo, java.sql.Connection.close()

9.7 Servlet

- **Evitare la sincronizzazione:**
  Per mantenere le caratteristiche di multi-threading di Servlet, evitare, se possibile, l'uso dello statement synchronized() all'interno di metodi della Servlet stessa
- **Uso del metodo init():**
  Utilizzare il metodo init() per tutte le attività di inizializzazione, le quali necessitano di essere eseguite una sola volta
- **Non usare la tag interface SingleThreadModel**:
  Non implementare javax.servlet.SingleThreadModel nelle servlet. Se una servlet implementa tale interfaccia il servlet container assicura che ciascuna istanza della servlet elabora una richiesta alla volta. I servlet container implementano questo comportamento mantenendo un pool di istanze di servlet e ridirezionando le richieste sulle istanze non utilizzate all'interno del pool. La SingleThreadModel interface fornisce un comodo metodo per l'accesso thread safe al metodo service(…), ma al costo di un aumento delle risorse all'aumentare delle richieste a cui deve rispondere il servlet container.

9.8 Disegno Pagine Web

Come nota generale sempre in ottica di aderenza al pattern architetturale, e' consigliabile limitare, il più possibile, logica di business nella parte di presentazione (vedi javascripts, applets), limitando i controlli a livello di numericita', data, obligatorietà.
E' consigliato l'uso dei fogli di stile (CSS) nel disegno delle pagine web. Questo al fine di minimizzare il numero di tag di formattazione (quindi alla dimensione fisica della pagina) e fornire un look uniforme all'intera applicazione.

9.9 Performance pagine Web

Evitare l'uso di componenti complessi/dinamici. Tenere in considerazione maggiormente il contenuto, spesso un componente graficamente elegante non da all'utente alcun contenuto informativo.

9.10 Documentazione codice

I commenti Java possono rientrare in due categorie: commenti di implementazione e commenti di documentazione. I commenti di implementazione seguono lo stesso formato del linguaggio C++, nello specifico rappresentati dalle sequenze `/*...*/` e `//`. I commenti di documentazione (conosciuti anche come "doc comments") sono peculiari del linguaggio Java, e sono delimitati dalla sequenza `/**...*/`. I commenti di documentazione vengono utilizzati dal tool javadoc per la creazione di documentazione di codice in formato standard HTML.

I commenti devono essere utilizzati per fornire informazioni sul codice non facilmente estrapolabili, e devono contenere esclusivamente note a questo relativo, al fine di facilitarne la lettura e la comprensione. In quest'ottica, informazioni concernenti, ad esempio, la struttura gerarchica di directory a cui fa riferimento un package, non devono essere incluse in commenti.

Discussioni relative a decisioni di disegno applicativo non ovvio possono, al fine di migliorare la comprensione di particolari scelte di codifica, essere incluse in commenti, purché non vi sia alcuna ridondanza di informazioni.

Evitare commenti che potrebbero essere soggetti ad incongruenze con una potenziale evoluzione del codice. In quest'ottica i commenti superflui riferiti a porzioni di codice auto esplicative non devono essere inclusi, considerando inoltre che una eccessiva presenza di commenti è di norma indice del basso livello qualitativo di codifica. In caso si trovi necessario commentare in maniera anomala porzioni di listato, dovrebbe essere presa in considerazione la possibilità di riscrivere il codice in dette porzioni per renderlo più chiaro.

I commenti non vanno inclusi in box delimitati da asterischi od altri caratteri, se non quelli indispensabili al relativo formato. Nei successivi paragrafi vengono esaminati sia i commenti di implementazione che di documentazione, approfondendo per ognuno di questi le relative sub-tipologie.

9.10.1 Commenti di Implementazione

I commenti di implementazione si dividono nei seguenti stili:

- blocco

  i commenti a blocco sono utilizzati per descrivere files, metodi, strutture dati ed algoritmi, e si estendono su molteplici linee. Tali commenti possono essere utilizzati all'inizio di ogni file e prima di ogni metodo, oltre che nel corpo di essi. Commenti a blocco interni a funzioni o metodi devono avere lo stesso livello di indentazione degli oggetti a cui si riferiscono.

  Un commento a blocco deve essere preceduto da una singola linea vuota (v. par. "Spaziatura", sez. "Linee Vuote").

```
/*
 * commento
 */
```

- linea singola

  i commenti a linea singola si riferiscono ad una singola linea di codice, e possono avere sia la stessa sintassi dei commenti a blocco, sia la sequenza iniziale `//`. Come i commenti a blocco, anche i commenti a linea singola devono essere preceduti da una linea vuota al fine di favorirne il distacco dal codice precedente, a meno che non appaiano accodati ad una linea di codice.

```
// commento a linea singola
metodoCalcolaSomma(a, b);

/* commento a linea singola */
metodoCalcolaSomma(a, b);

metodoCalcolaSomma(a, b);            /* commento a linea singola */
metodoCalcolaSomme(a, b);            // commento a linea singola
```

9.10.2 Commenti di Documentazione

I commenti di documentazione vengono utilizzati per estrapolare documentazione di codice direttamente dal listato sorgente attraverso l'utilizzo del tool javadoc. La documentazione così

generata sarà coerente con lo standard di documentazione Sun relativo alle "Java Platform API Specification", e quindi indipendente da eventuali formati proprietari di terze parti.

Non si intende riproporre qui informazioni relative all'utilizzo del tool javadoc, né approfondire il vasto numero dei tag disponibili, correlati dalla relativa sintassi, utilizzati per la creazione della documentazione di codice, ma ci si rivolge allo sviluppatore già familiare con l'utilizzo di tali strumenti, evidenziando quali tag e funzionalità siano obbligatoriamente da inserire all'interno di un sorgente Java affinché questo risulti compliant ai requirements espressi in questo testo.

Per ulteriori informazioni sui tools di documentazione ed al loro utilizzo si rimanda all'esauriente testo della nota tecnica Sun relativa all'utilità ed alle modalità di implementazione del tool javadoc, riportata nel par. "Riferimenti".

I tag richiesti in questo paragrafo non intendono limitare in alcun modo la volontà dello sviluppatore nell'integrare mezzi atti a fornire un livello di complessità di documentazione di codice superiore a quella richiesta. In quest'ottica lo sviluppatore è libero di inserire, in aggiunta ai tag qui descritti, ogni altra informazione ritenuta utile.

Struttura dei commenti di documentazione

Un commento di documentazione è costituito da due parti: una descrizione e zero o più tag. Queste due sezioni sono separate da un'unica riga contenente un singolo asterisco.

```
/**
 * Questa è la sezione di descrizione
 *
 * @tag1
 * @tag2
 */
```

- la prima linea deve avere un livello di indentazione uguale a quella del codice sottostante a cui fa riferimento, ed inizia con il simbolo `/**` seguito da un carattere di nuova riga.
- linee successive devono iniziare con un asterisco preceduto da un carattere di spaziatura, affinché tutti i caratteri asterisco risultino allineati verticalmente nell'ambito dello stesso commento.
- tag logicamente correlati devono essere separati in gruppi tramite l'inserimento di righe contenenti un singolo asterisco.
- l'ultima riga deve iniziare con la sequenza `*/` seguita da un carattere di nuova riga.

La prima frase di un commento di documentazione deve fornire un riassunto del codice a cui si riferisce, fornendone una descrizione tanto concisa quanto completa, poichè il tool javadoc utilizza tale frase iniziale per ogni membro, classe, interfaccia o package ai fini della relativa descrizione sintetica. Suddetta frase si intende terminata al primo carattere punto (`.`) seguto da un carattere di spaziatura, tabulatura, terminatore di riga, o al primo tag.

Linee guida generali nella scrittura di codice di documentazione

- omettere le parentesi per forme generiche di metodi e costruttori. Il metodo `add` può ad esempio avere due forme: `add(int, Object)` e `add(Object[])`. Se ci si vuole riferire ad entrambi le forme dello stesso metodo, evitare di referenziarlo come `add()`, poiché l'utilizzo di parentesi vuote potrebbe fuorviare il lettore, dando l'impressione di descrivere una terza forma del metodo che non accetta parametri. Includere in questo caso la parola 'metodo' a seguire, poiché il riferimento ad un nome di metodo senza parentesi potrebbe essere confuso con l'identificativo di un campo.
- scrivere documentazione di codice breve e sintetica.
- utilizzare la terza persona singolare nel corso delle descrizioni.
- evitare descrizioni che si limitino a ripetere i termini dell'oggetto a cui si riferiscono. E' doveroso espandere tali descrizioni in concetti che vadano oltre il nome dei campi, delle classi, delle interfacce o dei metodi che intendono descrivere.

  evitare il seguente esempio:

```
/**
 * Imposta il testo del bottone.
 *
 * @param testo       Testo del bottone.
 */
public void impostaTestoBottone(String testo) {
```

  utilizzare invece la seguente forma:

```
/**
 * Imposta il testo del bottone XYZ.
 *
 * @param testo       Il testo del bottone XYZ, In caso
 *                    tale testo sia null, al bottone XYZ
 *                    sara' assegnata la stringa di default
 *                    "strnigadidefault".
 */
    public void impostaTestoBottone(String testo) {
```

- non utilizzare acronimi non universalmente conosciuti, al contrario di URL o HTML.

Ordine dei tag

I seguenti tag di documentazione sono obbligatori, e vanno inclusi nel seguente ordine:

```
* @author             (classi ed interfacce)
* @version            (classi ed interfacce)
*
* @param              (metodi e costruttori)
* @return             (metodi)
* @exception   (@throws e' un sinonimo aggiunto in Javadoc 1.2)
* @see
```

Note:

- molteplici tag `@author` devono essere ordinati cronologicamente, con il creatore della relativa classe o interfaccia all'inizio della lista.
- molteplici tag `@param` devono essere ordinati per argomento.
- molteplici tag `@exception` devono essere inclusi in base all'ordine alfabetico delle relative eccezioni.

- molteplici tag `@see` devono seguire il seguente ordine:

```
@see #field
@see #Constructor(Type, Type...)
@see #Constructor(Type id, Type id...)
@see #method(Type, Type...)
@see #method(Type id, Type id...)
@see Class
@see Class#field
@see Class#Constructor(Type, Type...)
@see Class#Constructor(Type id, Type id...)
@see Class#method(Type, Type...)
@see Class#method(Type id, Type id...)
@see package.Class
@see package.Class#field
@see package.Class#Constructor(Type, Type...)
@see package.Class#Constructor(Type id, Type id...)
@see package.Class#method(Type, Type...)
@see package.Class#method(Type id, Type id...)
@see package
```

Tag obbligatori

I tag da includere obbligatoriamente nei commenti di documentazione sono i seguenti:

- `@author`

  specifica (opzionalmente) l'autore e (obbligatoriamente) la società fornitrice dell'oggetto a cui il commento fa riferimento.

- `@version`
- `@param`

  identifica il nome del parametro (non la tipologia), ed è seguito dalla descrizione dello stesso. Gli identificativi di parametro utilizzano, per convenzione, unicamente caratteri minuscoli. La descrizione di parametro inizia con un carattere minuscolo e non termina con un punto, a meno che non incorpori una o più frasi intere.

- `@return`

  omettere tale tag per metodi che ritornino `void`, includerlo invece in tutti gli altri casi, anche quando potrebbe apparire ridondante.

- `@exception/@throws`

  includere tale tag in riferimento ad ogni exception dichiarata dall'oggetto a cui il commento fa riferimento. Omettere tale tag per documentare una `NullPointerException`.

L'utilizzo del tag `@see` è opzionale, anche se altamente consigliato.

Per eventuali ulteriori dettagli necessari relativi alla stesura di commenti di documentazione per costruttori di default, exceptions, packages ed inner classes anonime, per l'inclusione di immagini, e per esempi di commenti di documentazione, si rimanda ai testi di riferimento citati in precedenza.

Per ulteriori informazioni riferirsi a :
http://java.sun.com/j2se/javadoc/writingdoccomments/index.html

9.11 Dichiarazioni

Seguono le linee guida da attuare in fase di dichiarazione di variabile, classe o interfaccia.

9.11.1 Numero di Dichiarazioni per Linea

Le dichiarazioni di variabili si devono limitare ad una per linea, evitando di accodarne molteplici, anche in presenza di più variabili dello stesso tipo. In ogni caso la spaziatura fra il tipo di variabile ed il relativo identificativo si deve limitare esclusivamente ad un unico carattere di spazio o tabulatura.

```
	int variabile1;

	int	variabile2;
	int	variabile3;
```

La sintassi riportata sopra è accettabile, mentre non lo è la seguente:

```
	int variabile1, variabile2, variabile3;
```

9.11.2 Inizializzazione

Le variabili devono essere inizializzate al momento della relativa dichiarazione. In caso in cui non sia possibile impostare un valore di default significativo iniziale, si renderà opportuno definire un valore nullo per la variabile coerente con la relativa tipologia.

9.11.3 Posizionamento

Le dichiarazioni vanno incluse esclusivamente all'inizio di ogni blocco di codice limitato da parentesi graffa. Nell'esempio qui riportato, una dichiarazione all'interno del blocco relativo allo statement `if` a seguito della chiamata al metodo `println()` è da riposizionare dopo la dichiarazione di variabile `b`.

```
	void metodo() {
		int a = 0;
		if (condizione) {
			int b = 0;
			System.out.println(...);
}
	}
```

L'unica eccezione a questa regola è rappresentata dallo statement `for`, in cui rimane possibile effettuare dichiarazioni all'interno dello statement stesso, come nel seguente esempio:

```
	for(int a = 0; a < n; a++) { ... }
```

Sono da evitare dichiarazioni che si sovrappongano a blocchi con scope di livello più alto, come nel seguente esempio:

```
	int count = 0;
	...
	metodo() {
		if (condizione) {
			int count = 0;
			...
		}
	}
```

9.11.4 Dichiarazioni di Classi o Interfacce

Le seguenti indicazioni vanno seguite in caso di dichiarazione di classe o interfaccia:

- non deve essere presente alcuno spazio fra il nome di un metodo e la parentesi tonda che ne determina l'inizio della lista dei parametri.
- la parentesi graffa aperta deve apparire alla fine della stessa riga dello statement di dichiarazione.

- la parentesi graffa chiusa deve risiedere su una propria linea e deve avere lo stesso livello di indentazione dell'espressione con cui la corrispondente parentesi graffa aperta segna l'inizio del relativo blocco di codice. In casi di blocchi nulli la parentesi graffa aperta deve essere immediatamente seguita dalla controparte chiusa.

```
class XYZ extends Object {
    int a;

    void metodo(int paramtero) {
        int b;
        ...
    }

    void metodovuoto() {}
}
```

9.12 Statements

Seguono le linee guida da attuare per ogni statement che caratterizza il linguaggio Java.

9.12.1 Statements Semplici

Ogni linea deve contenere esclusivamente un singolo statement. Parti di codice come quelle riportate nel seguente esempio devono essere assolutamente evitate:

```
        argv++; a+=4;
```

9.12.2 Compound Statements

I "compound statements" sono raggruppamenti di statement delimitati da parentesi graffe. Tali oggetti devono seguire le seguenti linee guida:

- gli statement facenti parte di un compound statement devono avere un livello di indentazione superiore.
- le parentesi graffa devono mantenere le caratteristiche definite per le dichiarazioni di classe o interfaccia riportate in precedenza.
- le parentesi graffa devono essere utilizzate per delimitare compound statements anche in caso racchiudano statement singoli, come in caso di una struttura `if-else` o `for`. Questa caratteristica migliora notevolmente la leggibilità del codice prodotto.

9.12.3 Statements Java

I seguenti paragrafi evidenziano la struttura degli statement del linguaggio Java. Ogni statement deve tassativamente rispettare la relativa struttura qui mostrata, includendo ogni carattere di spaziatura presente nell'ordine e nella quantità evidenziata. Nella struttura dei singoli statement il carattere di spaziatura singolo non può essere sostituito dal carattere di tabulatura. Per ulteriori informazioni, v. par. "Spaziatura", sez. "Caratteri di Spaziatura".

9.12.4 `return`

Lo statement `return` seguito da un valore non deve utilizzare parentesi a meno che non siano necessarie ad incrementare la leggibilità del codice, come riportato nei seguenti esempi:

```
return true;

return classe.metodo();

return (true ? 1 : 0);
```

9.12.5 if,if-else,if else-if else

La classe di statement `if-else` deve seguire la seguente forma:

```
if (condizione) {
    statements;
}

if (condizione) {
    statements;
} else {
    statements;
}

if (condizione) {
    statements;
} else if (condizione) {
    statements;
} else {
    statements;
}
```

E' da notare l'incondizionata presenza di parentesi graffa, anche in casi in cui l'`if` sia seguito da un singolo statement. Il seguente esempio rappresenta codice poco leggibile, e quindi non accettabile:

```
if (condizione)
    statement;
```

9.12.6 for

```
for (inizializzazione; condizione; update) {
    statement;
}
```

Uno statement `for` in cui ogni funzione possa essere inclusa nelle parentesi tonde deve avere la seguente forma:

```
for (inizializzazione; condizione; update);
```

In caso si utilizzi l'operatore `','` in inizializzazione o update evitare di utilizzare più di tre variabili.

9.12.7 while

```
while (condizione) {
    statement;
}

while (condizione);
```

9.12.8 do-while

```
do {
    statement;
} while (condizione);
```

9.12.9 `switch`

```
switch (condizione) {
case ABC:
    statements;
    /* commento */
case DEF:
    statements;
    break;
default:
    statements;
    break;
}
```

Va inserito un commento ogni volta che uno statement `case` non interrompa il flusso funzionale attraverso l'impiego di un `break`, continuando nella clausola `case` successiva.

Ogni statement `switch` deve includere una clausola `default`. Ogni clausola `default` all'interno di uno statement `switch` deve avere uno statement `break`, anche in casi in cui risulti ridondante.

9.12.10 `try-catch`

```
try {
    statements;
} catch (Exception e) {
    statements;
} finally {
    statements;
}
```

L'utilizzo della clausola `finally` in uno statement `try-catch` è opzionale.

9.13 Spaziatura

Mantenere convenzioni di spaziatura coerenti nell'ambito di molteplici sorgenti aumenta la leggibilità e la comprensione del codice in maniera sostanziale. Le seguenti linee guida relative a tale pratica vanno implementate incondizionatamente.

9.13.1 Linee Vuote

L'inclusione di linee vuote nel codice ne migliora la leggibilità, provvedendo ad isolarne logicamente i blocchi funzionalmente correlati fra loro.

Due linee vuote consecutive devono essere sempre utilizzate nelle seguenti circostanze:

- tra definizioni diverse di classi o interfacce nello stesso sorgente.
- tra sezioni di file non logicamente correlate fra loro.

Una sola linea vuota deve essere sempre utilizzata nelle seguenti circostanze:

- prima di ogni metodo.
- tra le dichiarazioni di variabili locali in un metodo ed il suo primo statement.
- prima di un commento .
- tra sezioni di codice logicamente separate all'interno dello stesso metodo.

9.13.2 Caratteri di Spaziatura

Un carattere di spaziatura deve essere utilizzato nelle seguenti circostanze:

- tra una keyword e una parentesi e prima delle parentesi graffa

```
while (true) {
    ...
}
```

- dopo una virgola

```
void metodo(int arg1, int arg2) {}
```

- tra un operatore ed i propri operandi, ad eccezione di `'.'`, ++ e –

```
a += c + d;
a =  (a + b) / (c * d);

while (d++ = s++) {
    n++;
}
println(“stringa” + “stringa”);
```

- tra le espressioni degli statement `for`

```
for (expr1; expr2; expr3);
```

- durante le operazioni di casting

```
metodo((byte) num, (Object) x);
metodo((int) (cp + 5), ((int) (i + 3)) + 1);
```

Nota:

Non devono essere presenti caratteri di spaziatura tra il nome di un metodo e la parentesi tonda che ne determina l’inizio degli argomenti, questo al fine di distinguere più facilmente fra metodi e keywords.

9.14 Invio mail

Le JavaMail api, forniscono un framework che permette alle applicazioni Java di interfacciare il servizio di mailing (trasmissione e ricezione) indipendentemente dalla piattaforma e dai protocolli utilizzati. Tali api sono fornite sia come package opzionale che come parte delle api J2EE Enterprise Edition. Proprio in riferimento a questo, nell’ambito di applicazioni enterprise l’invio di email deve utilizzare tali api in congiunzione dai servizi offerti dall’application server. Infatti è previsto che l’application server fornisca, dopo opportuna configurazione della risorsa, una mail session a seguito di una lookup sul servizio di naming. Questo consente di definire il server di posta (sia smpt che pop3/imap) a livello application server e rendere l’applicazione che ne fa uso indipendente dall’indirizzo fisico dello stesso.

Esempio di sorgente per invio di email:

```
import javax.mail.Message;
import javax.mail.Session;
import javax.mail.Transport;
import javax.mail.internet.InternetAddress;
import javax.mail.internet.MimeMessage;
import javax.naming.InitialContext;

public class MailSender
{
  public void inviaMail() throws DBContattiException
  {
    try
    {
      //recupero il naming context
```

```
      InitialContext ctx = new javax.naming.InitialContext();
      //recupero una mail session
      Session mailSession = (javax.mail.Session) ctx.lookup("java:comp/env/mail/mailsession");
      //preparo un messaggio da inviare con questa mail session
      MimeMessage msg = new MimeMessage(mail_session);
      //valorizzo il o i destinatari, il mittente, l'oggetto e il testo dell'mail
      msg.setRecipients(Message.RecipientType.TO, InternetAddress.parse("destinatario@tesoro.it"));
      msg.setFrom(new InternetAddress("mittente@tesoro.it"));
      msg.setSubject("Oggetto mail");
      msg.setText("Mail da J2EE Mail api");
      //invio la mail
      Transport.send(msg);
    }
    catch (Exception e)
    {
      logger.error("Errore nell'invio mail. " + e.getMessage());
    }
}
```

10 Supporto agli Sviluppatori di Applicazioni Web in ambiente Consip

10.1 Specifiche di accesso al DB2 e Oracle

Come già stato specificato nel capitolo relativo alle Linee guida di sviluppo riportiamo a titolo di esempio un estratto di codice per l'accesso attraverso l'utilizzo di un data source a un data base relazionale.

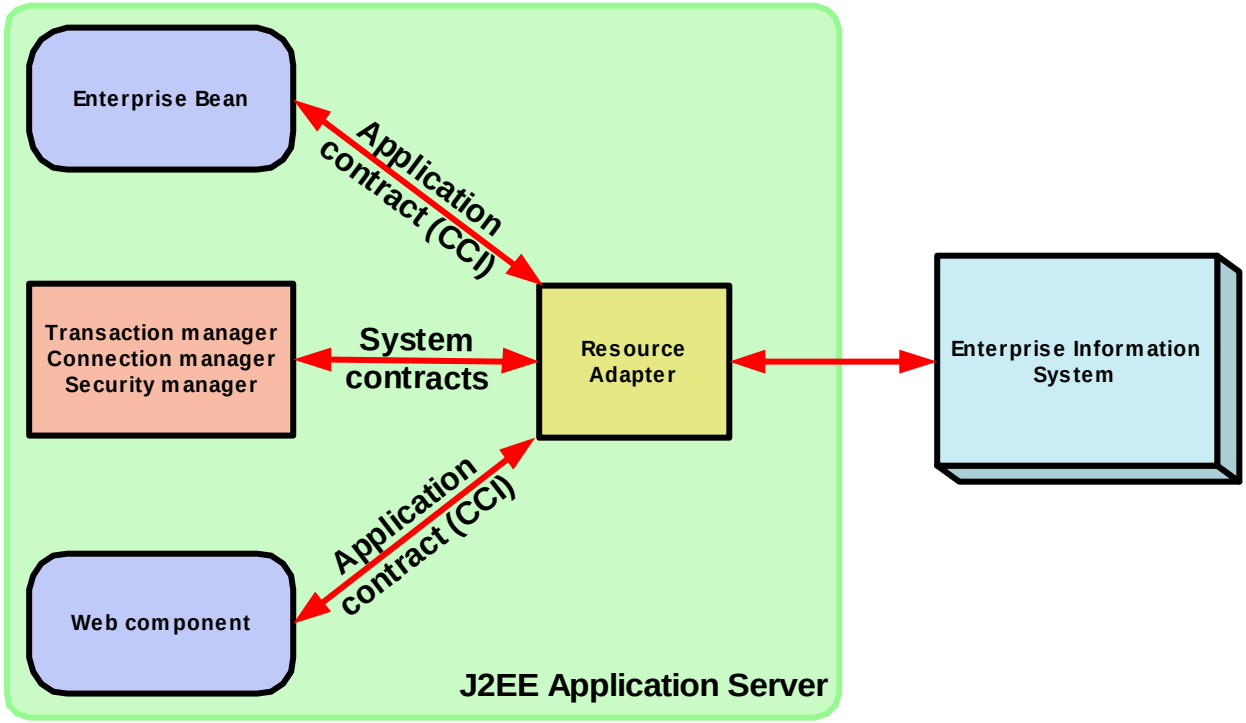
Esempio d'uso:

```
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import javax.sql.*;
import org.apache.log4j.Logger;
import org.apache.log4j.PropertyConfigurator;

public class EsempioDataSource
{
  //istanzio un logger per segnalare eventuali anomalie
  static Logger logger = Logger.getLogger(EsempioDataSource.class.getName());
  private DataSource pDs;

  public EsempioDataSource()
  {
    //configuro il logger
    PropertyConfigurator.configure("log4j.properties");
  }

  private boolean getDatasource()
  {
    try
    {
      //istanzio un InitialContext per ricavare attraverso il servizio di naming dirtectory
      //le risorse di cui necessito
      Context pContext = new InitialContext();
      //ricavo dal servizio di naming il datasource definito a livello di application server
      //in questo caso il datasource è stato definito con il nome 'Oracle9'
      pDs = (DataSource) pContext.lookup("java:comp/env/jdbc/Oracle9");
    }
    catch (NamingException e)
    {
      logger.error("Errore di naming " + e.getMessage());
      return false;
    }
    return true;
  }

  public ResultSet select(String pQuery) throws SQLException
  {
    ResultSet pResult = null;
    //chiamo la funzione che ricava il datasource dal servizio di naming
    getDatasource();

    logger.info("Query: " + pQuery);

    try
    {
      //ricavo la connessione dal datasource per effettuare la query
      Connection pConn = pDs.getConnection();
      //dalla conessione ricavo lo statement
      Statement pStatement = pConn.createStatement();
```

```
      //effettuo la query
      pResult = pStatement.executeQuery(pQuery);
    }
    catch(SQLException e)
    {
      logger.error("Errore SQL " + e.getMessage());
      throw e;
    }

    //ritorno il resultset relativo alla query effettuata
    return pResult;
  }
}
```

10.2 Specifiche di accesso alle transazione in ambiente CICS

La piattaforma J2EE include le specifiche di interfacciamento ai sistemi EIS, Enterprise Information System. Le specifiche definiscono una serie di interfacce che consentono ai produttori di tali sistemi di sviluppare e fornire componenti, **Resource Adapters**, che possono essere aggiunti a qualsiasi application server J2EE compliant. I componenti applicativi come ad esempio, EJB, servlet, Java beans, possono effettuare chiamate al resource adapter attraverso la **Client Common Interface** (Application contracts). Il resource adapter traduce queste chiamate in chiamate per l'EIS attraverso protocolli nativi come ad esempio ECI o EPI calls nel caso del CICS (System contracts).
Attraverso questo legame tra application server e resource adapter, indicato in figura come System contract, il resource adapter può utilizzare importanti risorse come il connection pooling, il transaction management (che consente di includere la chiamata verso l'EIS in una transazione XA) e il security manager che fornisci meccanismi di autenticazione, autorizzazione e criptazione della comunicazione tra application server e EIS.
Pertanto il produttore fornisce il resource adapter per l'EIS, mentre lo sviluppatore di applicazioni fa uso della CCI per interagire con il resource adapter e quindi con l'EIS.



Esempio d'uso:

```
/* Esempio di utilizzo della CCI per l'accesso al CICS attraverso una ECI call */

//J2EE CICS connector
import com.ibm.connector2.cics.*;
//J2EE Connector
import javax.namaing.Context;
import javax.resource.cci.*;
//...

//Funzione che chiama una transazione CICS
public String getCodiceFiscale(String nome, String congome)
{
  //istanzio un oggetto che serve a fornire le proprietà per la connessione, questo oggetto
  //sarà passato come argomento alla funzione getConnection della Connection Factory
  //in questo caso le proprietà che passiamo sono lle credenziali utente verso l'EIS
  ECIConnectionSpec eciConSpec = new ECIConnectionSpec("admin", "admin");

  //ricavo una connessione al CICS, prima ricavo una factory con una lookup JNDI, e poi chiamo
  //il metodo getConnection sulla factory
  Context nc = new InitialContext();
  ECIConnectionFactory eciFact =
          (ECIConnectionFactory)nc.lookup("java:comp/env/eis/ECIConnectionFactory");
  Conncection conn = eciFact.getConnection(eciConSpec);

  //istanzio un oggetto che serve ad impostare le proprietà dell'interazione con l'EIS
  ECIInteractionSpec eciIntSpec = new ECIInteractionSpec();
  //specifico il nome della transazione CICS da chiamare
  eciIntSpec.setFunctionName("TRANS");
  //specifico il tipo di interazione, in questo caso richiesta-risposta sincrona
  eciIntSpec.setIneractionVerb(InteractionSpec.SYNC_SEND_RECEIVE);

  //preparo i records che rappresentano le COMMAREA di input e output
  //questi oggetti implementano l'interfaccia javax.resource.cci.Record

  //Record di input
  AnagraficaKeyRecord anaKey = new AnagraficaKeyRecord();
  //Record di output
  AnagraficaRecord ana = new Anagrafica();

  anaKey.setNome(nome);
  anaKey.setCognome(cognome);

  //eseguo la transazione
  Interaction inter = conn.createInteraction();
  inter.execute(eciIntSpe, anaKey, ana);

  //chiudo la connessione
  conn.close();

  //ricavo il risultato dal record di output
  String codiceFiscale = ana.getCodiceFiscale();
  return codiceFiscale;
}
```

E' evidente che mentre la CCI fornisce il modello di base dell'interazione con un EIS, lo sviluppatore avrà bisogno di usare dei tools idonei che lo aiutino a generare gli oggetti Java che implementano l'interfaccia javax.resource.cci.Record e che rappresentano i legacy transaction metadata che devono essere scambiati con il sistema esterno. Nel nostro esempio tali oggetti non sono altro che le COPY dei programmi associati alla transazione richiamata.

11 Bibliografia

- [Gamma 1995]: E. Gamma, et al.: Design Patterns – Elements of Reusable Object-Oriented Software. Addison-Wesley.
- [Javasoft Guidelines] – http://java.sun.com/blueprints/guidelines
- [Javasoft J2EE] – http://java.sun.com/j2ee
- [J2EE Pattern] - http://java.sun.com/blueprints/patterns/j2ee_patterns/catalog.html

12 Note

Tutti gli esempi di codice sono riportati a solo titolo di esempio. Possono essere utilizzati, ma non si risponde di eventuali malfunzionamenti che dal loro uso possano derivare.