

Università degli Studi di Bologna



Facoltà di Ingegneria

Corso di Laurea in Ingegneria Informatica

S

Relazione

**Progettare e Sviluppare
Applicazioni Enterprise Distribuite
Multi-tier a Componenti
con Java™ 2 Platform, Enterprise Edition :
Servlet, JavaServer Pages
ed Enterprise JavaBeans,
architettura MVC e Application Server**

Studente

Fabio Rombaldoni

Ad Annalisa

Sommario

Capitolo 1	Introduzione.....	11
1.1	Le sfide dello sviluppo di Applicazioni Enterprise.....	12
1.1.1	Programmazione della produttività.....	12
1.1.2	Rapidità di risposta alla domanda (Response to Demand).....	12
1.1.3	Integrazione con gli ISs esistenti.....	13
1.1.4	Libertà di scelta tra server, tool e componenti.....	13
1.1.5	Mantenimento degli opportuni livelli di sicurezza.....	13
1.1.6	Riassumendo.....	14
1.2	La piattaforma Sun per soluzioni Enterprise.....	14
1.2.1	Una nota sulle architetture a componenti.....	16
1.2.2	Overview della piattaforma J2EE.....	17
1.2.2.1	Il modello multi-tier ed il J2EE Application Model.....	17
1.2.2.2	Container-Based Component Management.....	19
1.2.2.3	Supporto per componenti client.....	19
1.2.2.4	Supporto per componenti Business Logic.....	20
1.2.2.5	Supporto allo standard J2EE.....	20
1.2.3	I vantaggi della piattaforma J2EE.....	20
1.2.3.1	Sviluppo ed architettura semplificati.....	20
1.2.3.2	Facile scalabilità.....	21
1.2.3.3	Integrazione con gli EISs esistenti.....	21
1.2.3.4	Ampia scelta di server, tool e componenti.....	22
1.2.3.5	Modello di sicurezza semplificato ed unificato.....	22
1.3	Scenari possibili per Applicazioni J2EE.....	23
1.3.1	Scenario di Applicazione Multi-tier.....	24
1.3.2	Scenario di Client Stand-alone.....	24
1.3.3	Scenario di Applicazione Web-centric.....	25
1.3.4	Scenario di B2B.....	26
1.4	Packaging e Deployment di una Applicazione J2EE.....	26
1.4.1	Ruoli e Processi.....	27
1.4.2	Il Packaging di Applicazioni J2EE.....	29
1.4.3	I Deployment Descriptor.....	29
1.5	Riassunto.....	30
Capitolo 2	Le tecnologie della piattaforma J2EE.....	31
2.1	Component Technologies.....	31
2.1.1	Applet e Application Client.....	31
2.1.2	Componenti Web.....	31
2.1.2.1	La tecnologia Servlet.....	32
2.1.2.2	La tecnologia JavaServer Pages.....	32
2.1.2.3	Il Web Component Container.....	32
2.1.3	Componenti Enterprise JavaBeans.....	32
2.1.3.1	Session Bean.....	32
2.1.3.2	Entity Bean.....	33
2.1.3.3	L'EJB Component Container.....	33
2.1.4	Componenti, Container e Servizi.....	33
2.2	I Ruoli nella piattaforma J2EE.....	33
2.3	Service Technologies.....	34
2.3.1	Java DataBase Connectivity (JDBC).....	34
2.3.1.1	Stabilire una Connessione.....	34
2.3.1.2	Eeguire statement SQL e manipolare i risultati.....	35
2.3.1.3	Il Modello 2-tier.....	35
2.3.1.4	Il Modello 3-tier.....	35
2.3.1.5	JDBC e la piattaforma J2EE.....	36

2.3.2	Java Transaction API (JTA) e Java Transaction Service (JTS)	38
2.3.3	Java Naming and Directory Interface (JNDI)	38
2.3.4	J2EE Connector Architecture (JCA)	39
2.4	Communication Technologies	40
2.4.1	Protocolli Internet	40
2.4.2	Protocolli Remote Method Invocation (RMI)	43
2.4.3	Protocolli Object Management Group (OMG)	43
2.4.4	Tecnologie di Messaging	43
2.4.5	Data Formats	44
2.5	Riassunto	46
Capitolo 3	Un approfondimento sull'architettura MVC	47
3.1	Java e l'architettura MVC	48
3.2	L'architettura MVC e le applicazioni distribuite	49
3.3	Un primo esempio	50
3.4	Miglioriamo l'esempio: utilizzo delle tag library di JSP	54
Capitolo 4	Il Client Tier	59
4.1	I Client Web	59
4.2	I Client EJB	60
4.3	Design per molteplici tipi di Client	60
4.4	Considerazioni ulteriori sui Client J2EE	61
4.4.1	Client basati su pagine HTML	61
4.4.2	Client HTTP Content-Based	61
4.4.3	Client Intranet	62
4.4.4	Altri tipi di client	62
Capitolo 5	Il Web Tier	63
5.1	Applicazioni Web e Web container	64
5.2	Sviluppo di applicazioni Web e Creazione di contenuti dinamici	65
5.2.1	Common Gateway Interface (CGI)	65
5.2.2	ISAPI ed NSAPI	68
5.2.3	Active Server Pages (ASP)	68
5.2.4	Breve nota su JavaScript	69
5.2.5	Introduzione alle tecnologie Sun: Servlet e JavaServer Pages	69
5.2.5.1	La tecnologia Servlet	69
5.2.5.2	La tecnologia JavaServer Pages	69
5.3	Ruoli dei Web Component	70
5.4	Application Design	71
5.4.1	Applicazioni con Pagine JSP base e Servlet	71
5.4.2	Applicazioni con Componenti Modulari	72
5.4.3	Applicazioni EJB-Centric	73
5.5	Riassunto	73
Capitolo 6	La tecnologia Servlet	75
6.1	Overview	75
6.1.1	Che cos'è un Servlet?	75
6.1.2	Che cosa è un Servlet Container?	75
6.1.2.1	Nota sui Tipi MIME	75
6.1.3	Una sequenza di esempio	75
6.1.4	Confronto tra servlet ed altre tecnologie	76
6.2	L'interfaccia Servlet	76
6.2.1	Il package javax.servlet	77
6.2.2	Il package javax.servlet.http	78
6.2.3	Metodi per la gestione delle richieste	78
6.2.3.1	Metodi per gestire specifiche richieste HTTP	79
6.2.4	Numero di istanze: Servlet e multithreading	79
6.2.4.1	L'interfaccia SingleThreadModel	79

6.2.5	Il ciclo di vita di un Servlet.....	80
6.2.5.1	Caricamento ed Instanziamento	80
6.2.5.2	Inizializzazione	81
6.2.5.3	Gestione delle richieste	81
6.2.5.4	Fine del servizio	81
6.2.6	Un primo esempio di classe Servlet	82
6.2.7	Una nota sul metodo service	82
6.3	L'interfaccia ServletContext.....	83
6.4	La Richiesta	84
6.4.1	I parametri del protocollo HTTP	85
6.4.2	Attributi	85
6.4.3	Header HTTP.....	85
6.4.4	I Cookies.....	85
6.4.4.1	Perché i cookie? I limiti del protocollo HTTP.....	86
6.4.4.2	Manipolare cookies con i Servlet.....	86
6.4.4.3	Un esempio completo	87
6.4.5	Attributi SSL	87
6.5	La Risposta	88
6.5.1	Buffering	89
6.5.2	Header HTTP.....	89
6.5.3	Notificare errori utilizzando Servlet.....	90
6.5.4	Chiusura di un oggetto Response	90
6.6	Sessioni	90
6.6.1	Meccanismi per il Session Tracking.....	91
6.6.2	Cookie vs. Sessioni Utente	91
6.6.3	Sessioni dal punto di vista di un Servlet	91
6.6.4	La classe HttpSession.....	92
6.6.5	Un esempio di gestione di una Sessione Utente.....	93
6.6.6	Durata di una Sessione Utente	93
6.6.7	URL Rewriting	94
Capitolo 7	La tecnologia JavaServer Pages.....	95
7.1	Overview	95
7.1.1	Concetti generali	95
7.1.2	Che cosa è una pagina JSP?	96
7.1.3	Vantaggi di JSP	96
7.1.4	Applicazione Web	96
7.1.5	Confronto con altre tecnologie	96
7.1.6	JavaServer Pages vs. Servlet.....	97
7.1.7	Un primo esempio	98
7.2	Elementi di una pagina JSP	100
7.2.1	Direttive.....	100
7.2.1.1	La direttiva page	100
7.2.1.2	La direttiva include	101
7.2.1.3	La direttiva taglib	101
7.2.2	Elementi di scripting	102
7.2.3	Action	103
7.3	Commenti	104
7.4	Gestione degli errori nelle pagine JSP.....	104
7.4.1	Errori al momento della compilazione	104
7.4.2	Errori al momento della richiesta	104
7.4.3	Creazione e uso di una pagina di errore	104
7.5	Oggetti e Scope.....	105
7.6	Oggetti Impliciti.....	105
7.6.1	L'oggetto request	106
7.6.2	L'oggetto response	106
7.6.3	L'oggetto out	107
7.6.4	L'oggetto pageContext	107

7.6.5	L'oggetto config.....	107
7.6.6	L'oggetto exception.....	107
7.6.7	L'oggetto session.....	107
7.6.7.1	Memorizzare i dati nell'oggetto session.....	107
7.6.7.2	Leggere il contenuto di una variabile di sessione.....	107
7.6.7.3	Altri metodi dell'oggetto session.....	108
7.6.8	L'oggetto application.....	108
7.7	Utilizzo dei JavaBeans	108
7.7.1	Il modello JavaBeans e le sue caratteristiche fondamentali.....	109
7.7.2	Proprietà dei bean.....	109
7.7.3	Aggiunta di un bean in una pagina JSP.....	110
7.7.4	Un esempio di bean.....	110
7.8	Breve accenno alle Tag Library	112
Capitolo 8	L'Enterprise JavaBeans Tier.....	113
8.1	La Business Logic.....	113
8.1.1	Requisiti comuni dei Business Object.....	114
8.2	Enterprise Bean come J2EE Business Object.....	115
8.2.1	L'architettura EJB.....	116
8.2.2	Enterprise Bean e EJB Container.....	116
8.2.2.1	La Home Interface.....	117
8.2.2.2	La Remote Interface.....	118
8.2.2.3	La Enterprise Bean Class.....	118
8.2.2.4	Introduzione a Entity Bean e Session Bean.....	118
8.3	Entity Bean	119
8.3.1	Linee guida all'uso degli Entity Bean.....	119
8.3.2	Esempio: uno User Account Bean.....	119
8.3.3	La persistenza dei dati.....	120
8.4	Session Bean.....	120
8.4.1	Stateful Session Bean.....	121
8.4.1.1	Esempio: uno Shopping Cart Bean.....	121
8.4.2	Stateless Session Bean.....	122
8.4.2.1	Esempio: un Catalog Bean.....	122
8.5	Riassunto.....	123
Capitolo 9	L'Enterprise Information System Tier.....	125
9.1	Una applicazione Internet E-Store	126
9.2	Una applicazione Intranet Human Resources.....	126
9.3	Una applicazione per il Distributed Purchasing.....	127
Capitolo 10	Un approfondimento sugli Application Server.....	129
10.1	Gli elementi essenziali di un Application Server.....	130
10.2	Java e i server, una coppia ideale	131
10.3	Perché l'Application Server? Vantaggi	132
10.4	Uno sguardo al mercato: una confusione voluta?.....	133
10.5	Valutazione degli Application Server: guida alla scelta	135
10.6	Verso l'azienda elettronica.....	137
Appendice	Programmazione Object Oriented e Java	139
A.1	Introduzione al paradigma Object Oriented.....	139
A.1.1	Ereditarietà.....	139
A.1.2	Incapsulamento.....	140
A.1.3	Polimorfismo.....	141
A.2	Introduzione a Java.....	141
A.2.1	La magia di Java: i bytecode.....	141
A.2.2	Le caratteristiche essenziali di Java.....	142
A.2.2.1	Indipendenza dalla piattaforma.....	142
A.2.2.2	Uso della memoria e multithreading.....	143

A.2.3	Tipi primitivi, strutture sintattiche e controllo di flusso	143
A.2.4	Metodi	145
A.2.5	Classi	145
A.2.6	Package Java	145
A.2.7	Il modificatore public	145
A.2.8	L'istruzione import.....	146
A.3	Incapsulamento in Java	146
A.3.1	Modificatori public, private e protected	146
A.3.2	L'operatore new	147
A.3.3	Costruttori.....	147
A.4	Ereditarietà in Java.....	148
A.4.1	Disegnare una classe base	148
A.4.2	Overload di metodi	148
A.4.3	Estendere una classe base	149
A.4.4	Ereditarietà ed incapsulamento.....	149
A.4.5	Ereditarietà e costruttori.....	149
A.4.6	Overriding di metodi	149
A.4.7	L'oggetto Object.....	149
A.5	Accenno alle Eccezioni.....	150
A.6	Polimorfismo ed ereditarietà avanzata	150
A.6.1	Polimorfismo : "un'interfaccia, molti metodi"	150
A.6.2	Interfacce.....	151
A.6.3	Classi astratte.....	151
A.7	Java Thread	152
A.7.1	Thread di sistema	153
A.7.2	La classe java.lang.Thread	153
A.7.3	L'interfaccia Runnable	154
Bibliografia	157
	Documenti principali	157
	Altri documenti.....	157

Capitolo 1 Introduzione

Internet ed il World Wide Web rappresentano le fondamenta sulle quali le aziende stanno lavorando per costruire un **Information Economy**. In questa economia, le informazioni assumono lo stesso valore di beni e servizi, e divengono una parte vitale del mercato. L'Information Economy sfida le aziende odierne a ripensare radicalmente al loro modo di fare business.

Le aziende hanno da sempre tentato di guadagnare un vantaggio competitivo con qualsiasi ragionevole mezzo a loro disposizione, incluse le più recenti tecnologie. Questo è un istinto di sopravvivenza naturale: qualsiasi organizzazione (dalle società per azioni alle organizzazioni non-profit alle istituzioni governative) è continuamente alla ricerca di modi per rimanere al passo con i tempi ed adattarsi ai cambiamenti in atto; gli specialisti dell'Information Technology lavorano per trasformare le nuove sfide in business di successo.

Nell'Information Economy le informazioni rivestono un valore strategico per un'organizzazione e l'abilità nel capitalizzare questo valore diviene la chiave del successo. Ormai da qualche tempo i professionisti dell'IT devono affrontare una sfida: la domanda per un "responsive management of information assets" (gestione/controllo sensibile delle qualità delle informazioni).

La risposta iniziale fu quella di assicurare che tutte le "critical business functions" (funzioni critiche del business) fossero efficacemente gestite da sistemi computerizzati. Più recentemente, gli sforzi si sono concentrati sull'integrazione tra questi sistemi e sulla correlazione di dati provenienti dalle fonti più disparate in informazioni che giovino a specifiche necessità strategiche. Fusioni di enti pubblici ed aziende private, acquisizioni e partnership commerciali sono state ulteriori incentivi per le organizzazioni ad integrare tali informazioni.

Le **applicazioni distribuite** sono i pacchetti nei quali un'organizzazione distribuisce informazioni così come distribuisce un prodotto. Esse aggiungono ed estraggono valore dagli information assets di un'organizzazione; permettono alle organizzazioni IT di focalizzare la loro attenzione su specifiche funzionalità per specifiche necessità dell'utente. Rendendo disponibili le informazioni all'interno di un'organizzazione, esse aggiungono valore strategico ai processi di gestione e pianificazione. Proiettando in maniera mirata e selettiva gli information assets all'esterno dell'organizzazione, esse permettono scambi di informazioni che sarebbero reciprocamente costosi tra clienti, fornitori ed organizzazione stessa.

Nello scenario competitivo dell'Information Economy, i **tempi di risposta** sono la chiave del valore delle applicazioni enterprise. Le organizzazioni richiedono sviluppo e deployment delle applicazioni in tempi brevi, e possibilità di raffinare ed accrescere facilmente tali applicazioni per incrementarne il valore. Inoltre hanno bisogno di integrare queste applicazioni con i sistemi informativi (Enterprise Information Systems o EISs) esistenti in maniera facile ed efficiente, e di scaricarle senza sforzi alcuni per venire incontro a cambiamenti di domanda.

Lo scopo di **Java™ 2 Platform, Enterprise Edition** (J2EE™ Platform) è di definire uno standard di funzionalità che aiutino ad affrontare queste sfide così da incrementare la competitività della azienda nell'Information Economy. La piattaforma J2EE supporta applicazioni distribuite che traggono vantaggio da un ampio insieme di tecnologie nuove ed in evoluzione, semplificando allo stesso tempo lo sviluppo attraverso un component-based application model (modello di applicazione basata su componenti). Il modello J2EE supporta applicazioni che vanno dalle tradizionali applicazioni client/server distribuite sulle Intranet aziendali agli e-commerce Web site su Internet. La piattaforma J2EE fornisce uno standard singolo e consolidato che accresce l'opportunità delle aziende di proiettare i loro sistemi informativi oltre i loro confini storici, evitando allo stesso tempo i rischi impliciti di tale processo.

Con la recente esplosione degli Application Server e con la nascita delle specifiche J2EE oggi si può quindi disporre di una serie di potenti mezzi per realizzare applicativi di e-business complessi in maniera semplice e con possibilità di portabilità, scalabilità e facile manutenzione del software in questione. Secondo molti esperti la chiave per il successo oggi è creare applicazioni compatibili con J2EE.

Uno dei principali vantaggi offerti dalla tecnologia J2EE è la **portabilità** delle applicazioni. Per creare applicazioni portabili basate sulla tecnologia J2EE, gli sviluppatori devono scrivere componenti applicativi conformi alle specifiche J2EE e testarli al fine di verificarne la conformità. Questo processo contribuisce a garantire che l'applicazione funzionerà su qualsiasi Application Server certificato per la tecnologia J2EE, indipendentemente dalla piattaforma. La tecnologia J2EE

consente inoltre di gestire molte funzioni comuni quali gestione delle risorse, transazioni, sicurezza e autenticazione, persistenza, servizi di naming e di directory e servizi di messaging. Quindi è possibile concentrarsi sulla risoluzione dei problemi strettamente connessi al business, lasciando i dettagli della programmazione di livello inferiore all'architettura.

1.1 Le sfide dello sviluppo di Applicazioni Enterprise

Mentre il timing, cioè la collocazione nel tempo, è sempre stato un fattore critico nell'adottare nuove tecnologie, l'accelerazione inerente ad un modello di business virtuale information-driven (guidato dalle informazioni) ha posto sempre più grande enfasi sui tempi di risposta. In una economia influenzata da Internet, è imperativo non solo proiettare i sistemi aziendali su svariati canali, ma farlo ripetutamente e tempestivamente, con frequenti aggiornamenti sia alle informazioni sia ai servizi.

La sfida principale è dunque quella di non sfigurare davanti all'andatura iper-competitiva di Internet (la Rete delle reti) mantenendo e facendo leva sul valore dei sistemi aziendali esistenti. In questo scenario, la tempestività è assolutamente critica per guadagnare e mantenere un margine competitivo. Diversi fattori possono accrescere o ostacolare l'abilità di un'organizzazione nel distribuire rapidamente applicazioni enterprise custom e nel massimizzarne il valore durante il ciclo di vita (lifetime):

- Programmazione della produttività
- Rapidità di risposta alla domanda (Response to Demand)
- Integrazione con gli ISs esistenti
- Libertà di scelta tra server, tool e componenti
- Mantenimento degli opportuni livelli di sicurezza

1.1.1 Programmazione della produttività

L'abilità nello sviluppo e deployment delle applicazioni è la chiave del successo nell'Information Economy. Le applicazioni devono passare velocemente da prototipo a produzione, e continuare ad evolvere anche dopo essere state deployate. La produttività diviene in tal modo vitale. Fornire team di sviluppo dotati di modalità standard di accedere ai servizi richiesti dalle applicazioni multi-tier e di supportare una molteplicità di client può contribuire sia al miglioramento sia alla flessibilità.

Un fattore destabilizzante per Internet e altre applicazioni di Enterprise Computing (EC) e l'attuale **divergenza tra tecnologie e modelli di programmazione**. Storicamente (in termini Web), tecnologie come HTML e CGI hanno fornito un meccanismo per distribuire contenuti dinamici, mentre sistemi back-end come Transaction Processor e DBMS hanno fornito accessi controllati ai dati che dovevano essere presentati e manipolati. Queste tecnologie presentano diversi modelli di programmazione, alcuni basati su standard ben definiti, altri su standard ad-hoc, altri ancora su architetture proprietarie.

Mancando un unico modello di applicazione (application model), può risultare difficile per i team comunicare requisiti applicativi in maniera efficace e produttiva; la progettazione di applicazioni risulta così più complessa. Inoltre gli skill set richiesti per integrare queste tecnologie non sono ben organizzati per un'efficace divisione del lavoro. Per esempio, lo sviluppo di CGI richiede ai programmatori di definire sia il contenuto sia il layout di una pagina Web dinamica.

Un altro fattore che complica lo sviluppo di applicazioni è la scelta dei client. Mentre molte applicazioni possono essere distribuite a browser Web attraverso HTML statico o generato dinamicamente, altre possono aver bisogno di supportare uno specifico tipo di client o diversi tipi di client contemporaneamente. Il modello di programmazione deve prevedere il supporto ad una molteplicità di configurazioni client, con il minimo effetto sull'architettura base o sulla business logic centrale dell'applicazione.

1.1.2 Rapidità di risposta alla domanda (Response to Demand)

Si immagini che un'impresa di costruzione edilizia (brick&mortar=mattone e malta) tenti di incrementare la sua clientela base di un fattore 10. Quanto tempo e quanti sforzi si dovranno

spendere per rimodellare i depositi, costruire nuovi magazzini, e così via, per essere all'altezza dell'obiettivo prefissatosi? Il fatto è che il costante riammodernamento impatterà drasticamente sulla capacità di servire i clienti che l'impresa sta tentando di attrarre. Anche questo proietta l'azienda nell'Information Economy. La capacità delle applicazioni di scalare facilmente e di adattarsi automaticamente ad una crescita più o meno prevista è la chiave per raggiungere gli obiettivi. I sistemi che richiedono una qualsiasi ristrutturazione o redeployment per scalare impediranno o rallenteranno la crescita e diminuiranno la performance prevista dalla compagnia.

Al fine di scalare efficacemente, i sistemi devono essere progettati per gestire con facilità interazioni con molteplici client. Tali sistemi richiedono meccanismi per una gestione efficiente delle risorse di sistema e dei servizi come connessioni a database e transazioni. Devono avere accesso a caratteristiche quali bilanciamento automatico del carico (automatic load balancing = distribuire i carichi su vari server), senza alcuno sforzo sullo sviluppatore. Infine le applicazioni dovrebbero poter girare su qualsiasi server appropriato e cambiare facilmente configurazioni server qualora ce ne fosse bisogno.

1.1.3 Integrazione con gli ISs esistenti

Gran parte dei dati di valore per un'organizzazione sono stati raccolti negli anni su IS esistenti. Gran parte dell'investimento in programmi risiede in applicazioni su quegli stessi sistemi. La sfida per gli sviluppatori di applicazioni enterprise è come riutilizzare ed amplificare questo valore.

Per raggiungere quest'obiettivo, gli sviluppatori hanno bisogno di modalità standard per accedere ai servizi del Middle Tier e di back-end quali DBMS e Transaction Monitor. Inoltre hanno bisogno di sistemi che forniscano questi servizi in maniera consistente, così che non siano richiesti nuovi stili o modelli di programmazione nel momento in cui l'integrazione si espande per includere sistemi diversi all'interno di un'azienda.

1.1.4 Libertà di scelta tra server, tool e componenti

Lo sviluppo di applicazioni "responsive" richiede l'abilità di unire ed armonizzare soluzioni per pervenire alla configurazione ottima per il processo che si sta considerando. Ciò è semplificato dalla libertà di scelta di server, tool e componenti.

La possibilità di scegliere tra diversi prodotti server permette alle organizzazioni di selezionare configurazioni fatte su misura per i requisiti delle loro applicazioni. Inoltre permette di muoversi velocemente e facilmente da una configurazione all'altra non appena la domanda interna ed esterna lo richiedano.

I team di sviluppo dovrebbero poter adottare nuovi strumenti al bisogno. Oltre a ciò, ciascun membro di un team dovrebbe avere accesso ai tool più appropriati alla propria competenza e al proprio contributo.

Infine, gli sviluppatori possono trarre vantaggio da competenze esterne e migliorare la loro produttività se possono rivolgersi ad un mercato che mette a disposizione un'ampia scelta di componenti applicativi.

1.1.5 Mantenimento degli opportuni livelli di sicurezza

Può apparire in qualche modo ironico, ma proiettare gli information assets per estrarre il loro valore può mettere in pericolo quello stesso valore.

Tradizionalmente, i settori IT hanno mantenuto un livello relativamente alto di controllo sugli ambienti di esecuzione sia dei server sia dei client. Quando gli information assets sono proiettati in ambienti meno protetti (es. Internet), è decisamente importante mantenere altissimi livelli di sicurezza sugli assets più sensibili, ed allo stesso tempo permettere accesso apparentemente senza ostacoli agli altri.

Una delle difficoltà nell'integrare sistemi diversi è fornire un modello di sicurezza unificato. La sicurezza deve essere compatibile con i meccanismi esistenti. Nei casi in cui gli utenti devono accedere ad informazioni sicure, i meccanismi devono mantenere alta sicurezza, affidabilità e user confidence, e rimanere allo stesso tempo il più possibile riservati e trasparenti.

1.1.6 Riassumendo...

Riassumendo, il problema riguarda la realizzazione di applicazioni distribuite soddisfacendo i seguenti requisiti:

- Riduzione dei tempi di sviluppo
- Definizione di un modello di progettazione standard
- Supporto per applicazioni client eterogenee
- Scalabilità
- Efficiente gestione delle risorse
- Gestione del carico di lavoro automatica
- Integrazione con sistemi preesistenti
- Ampia scelta di server, strumenti e componenti
- Sicurezza

1.2 La piattaforma Sun per soluzioni Enterprise

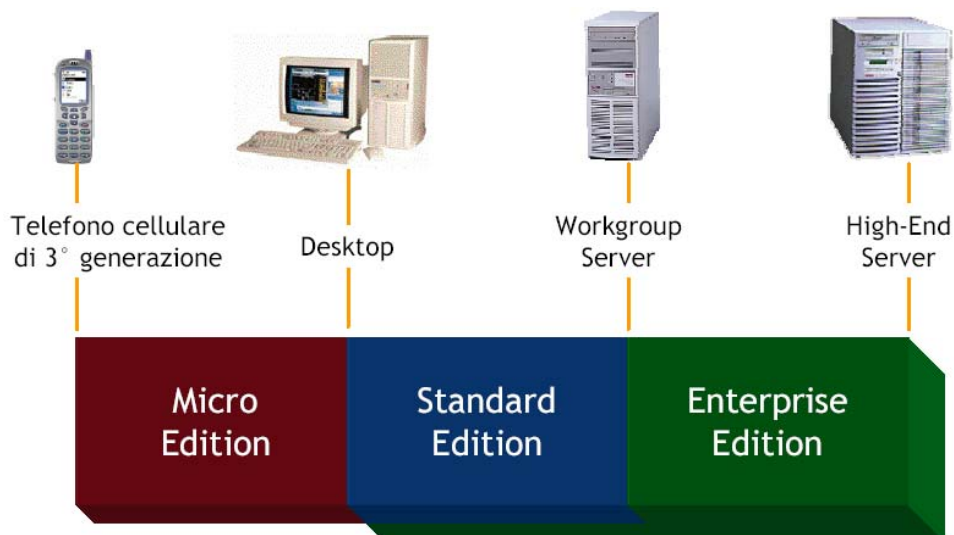


Figura 1.1 Le piattaforme Sun

Da quanto emerso in precedenza, oggi le aziende hanno bisogno di estendere la loro portata, il loro raggio d'azione e la loro visibilità, ridurre i costi ed abbassare i tempi di risposta, fornendo servizi easy-to-access ai loro clienti, partner, impiegati e fornitori.

Tipicamente, le applicazioni che forniscono tali servizi devono combinare gli EISs esistenti con nuove business function che distribuiscono servizi ad un largo insieme di utenti. Questi servizi devono essere:

- ✘ **Altamente disponibili**, per venire incontro alle continue e pressanti richieste degli ambienti della moderna New Economy.
- ✘ **Sicuri**, per proteggere la privacy degli utenti e l'integrità dei dati aziendali.
- ✘ **Affidabili e scalabili**, per assicurare che le transazioni aziendali siano accuratamente e prontamente elaborate.

Per diversi motivi, questi servizi sono generalmente implementati come applicazioni distribuite consistenti di diversi tier (multi-tier), le quali includono client sul front-end, fonti dati sul back-end, e uno o più Middle Tier tra loro dove è fatta la gran parte del lavoro di sviluppo. Il Middle Tier implementa i nuovi servizi che integrano gli EISs esistenti con le nuove business function ed i dati messi a disposizione appunto dai nuovi servizi. Il Middle Tier protegge e nasconde al Client Tier la complessità dell'azienda e trae vantaggio dalla rapida maturazione delle tecnologie legate ad Internet per minimizzare l'amministrazione e la formazione dell'utente.

La piattaforma J2EE riduce i costi e la complessità dello sviluppo di questi servizi multi-tier, permettendo l'implementazione di servizi che possono essere rapidamente deployati e facilmente estesi nel momento in cui l'azienda dovesse rispondere a pressioni competitive.

La piattaforma J2EE è quindi la soluzione proposta da **Sun Microsystems, Inc.** per l'EC, ed ottiene i vantaggi precedentemente accennati definendo un'architettura standard che si compone dei seguenti elementi principali:

1. J2EE Specification
2. J2EE Reference Implementation
3. J2EE Compatibility Test Suite
4. J2EE BluePrints (linee guida alla progettazione)

Le **Specification** elencano gli elementi necessari alla piattaforma e le procedure da seguire per una corretta implementazione con J2EE, definendo come la piattaforma (specificata da un insieme di APIs e di linee di condotta) lavora.

Specifiche	Sigla	Versione	Indirizzo
Java™ 2 Platform, Enterprise Edition	J2EE	1.3	http://java.sun.com/j2ee/docs.html
Java™ 2 Platform, Standard Edition	J2SE	1.3	http://java.sun.com/j2se/1.3/docs/api/index.html
Enterprise JavaBeans™	EJB	2.0	http://java.sun.com/products/ejb
JavaServer Pages™	JSP	1.2	http://java.sun.com/products/jsp
Java™ Servlet		2.3	http://java.sun.com/products/servlet
Java DataBase Connectivity™	JDBC	3.0	http://java.sun.com/products/jdbc
Java™ Naming and Directory Interface	JNDI	1.2	http://java.sun.com/products/jndi
Java™ Message Service	JMS	1.0.2	http://java.sun.com/products/jms
Java™ Transaction API	JTA	1.0.1	http://java.sun.com/products/jta
Java™ Transaction Service	JTS	1.0	http://java.sun.com/products/jts
JavaMail™ API		1.1	http://java.sun.com/products/javamail
JavaBeans™ Activation Framework	JAF	1.0	http://java.sun.com/beans/glasgow/jaf.html
Java™ IDL	IDL		http://java.sun.com/j2se/1.3/docs/guide/idl/index.html
RMI over IIOP	RMI-IIOP	1.0.1	http://java.sun.com/j2se/1.3/docs/guide/rmi-iiop/index.html
J2EE™ Connector Architecture	JCA	1.0	http://java.sun.com/j2ee/connector
Java API for XML Parsing	JAXP	1.0	http://java.sun.com/xml
Java™ Authentication and Authorization Service	JAAS	1.0	http://java.sun.com/products/jaas

Tabella 1.1 Le specifiche relative alle tecnologie della piattaforma J2EE

La **Reference Implementation** contiene prototipi che rappresentano istanze semanticamente corrette della piattaforma J2EE al fine di fornire all'industria del software modelli completi per test e confronti. Include tool per lo sviluppo, il deployment e l'amministrazione, un J2EE Application Server, un Web server, un database relazionale, le J2EE APIs.

La Reference Implementation risponde a diverse funzioni. In primo luogo rappresenta la **definizione operativa** della piattaforma J2EE. Sotto tale veste è usata dai vendor come "goal standard" della piattaforma J2EE per determinare che cosa la loro implementazione della piattaforma deve fare sotto un particolare insieme di condizioni applicative. E' anche usata come piattaforma standard per eseguire la J2EE Compatibility Test Suite. La sua funzione secondaria, ma più evidente, è quella di piattaforma **disponibile gratuitamente** per divulgare J2EE. Sebbene non sia un prodotto commerciale ed i termini della sua licenza ne proibiscano l'uso commerciale, essa è disponibile gratuitamente in due formati (binary o source) per dimostrazioni, prototyping e ricerca accademica. E' possibile scaricare il **J2EE SDK** (Software Development Kit) all'indirizzo seguente: <http://java.sun.com/j2ee/download.html#sdk>.

La **Compatibility Test Suite** è una suite di test di compatibilità, fornita da Sun per verificare che un prodotto J2EE di un vendor sia stato implementato compatibilmente con lo standard così da assicurarne la portabilità (descritta dallo slogan "Write Once, Run Anywhere" o WORA). Il vendor farà il deployment, configurerà ed eseguirà questa suite di test sulla sua piattaforma. La suite include test per assicurarsi che le APIs di J2EE siano state implementate. I test verificheranno che le tecnologie componenti di J2EE sono disponibili e lavorano insieme opportunamente. La suite

inoltre include un insieme di applicazioni J2EE ufficiali di esempio per verificare che tutte le piattaforme siano capaci di deployarle ed eseguirle in modo consistente.

Le **BluePrints** definiscono un modello per la progettazione e la programmazione di applicazioni multi-tier thin-client, basato sulla metodologia best-practice per favorire un approccio ottimale alla piattaforma. Guidano il programmatore analizzando quanto va fatto e quanto no con J2EE, e forniscono le basi della metodologia legata allo sviluppo di sistemi multi-tier con J2EE.

La piattaforma J2EE rappresenta quindi uno standard unico per l'implementazione ed il deployment di applicazioni enterprise multi-tier a componenti. La piattaforma è stata disegnata attraverso un **open process**, cui hanno preso parte diversi vendor, al fine di garantire conformità al più ampio insieme di requisiti per applicazioni enterprise.

A conferma di ciò, l'ultima versione delle specifiche della piattaforma J2EE (1.3) è stata creata sotto il Java Community Process come JSR-058. Il JSR-058 Expert Group include rappresentanti delle seguenti compagnie ed organizzazioni operanti nel mondo IT: Allaire, BEA Systems, Bluestone Software, Borland, Bull S.A., Exoffice, Fujitsu Limited, GemStone Systems, Inc., IBM, Inline Software, IONA Technologies, iPlanet, jGuru.com, Orion Application Server, Persistence, POET Software, SilverStream, Sun, e Sybase.

La Figura 1.2 mostra un'ipotetica architettura distribuita per l'Enterprise Computing (=calcolo eseguito da un gruppo di programmi interagenti attraverso una rete) e pone particolare attenzione su alcune delle tecnologie componenti la piattaforma J2EE e sulle loro modalità di interconnessione.

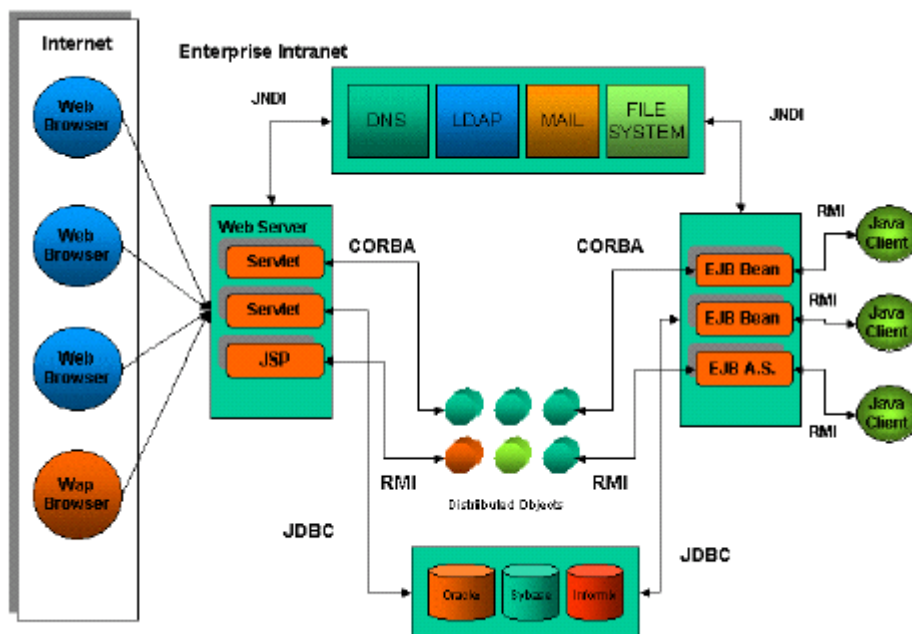


Figura 1.2 Un'ipotetica architettura distribuita per l'Enterprise Computing

1.2.1 Una nota sulle architetture a componenti

Un'architettura a componenti è tale se fornisce un framework per il riutilizzo e l'integrazione del software, se definisce uno standard aperto ed un set di servizi integrati nel framework e, soprattutto, se fornisce un Component Execution Environment (ambiente di esecuzione dei componenti) che gestisce la vita dei componenti, la loro replica, la suddivisione ed il bilanciamento automatico del carico, il fail-over (per gestire l'esecuzione di applicazioni con alti volumi di transazioni), il clustering, l'integrazione con i servizi.

J2EE è sicuramente la piattaforma che meglio ha incarnato queste caratteristiche affermandosi di conseguenza come l'architettura a componenti di riferimento per le applicazioni enterprise e per quelle Web mission-critical. Ovviamente non è l'unica architettura a componenti esistente: Microsoft ha rilasciato la sua nuova piattaforma **.NET** e l'OMG ha ultimato la definizione del Corba Component Model (**CCM**).

Le architetture a componenti sono la migliore soluzione per realizzare applicazioni transazionali e scalabili che integrano in modo stretto le risorse aziendali. Nella piattaforma J2EE questo è reso possibile dall'integrazione di una suite di servizi ed interfacce standard per la ricerca dei componenti (JNDI), per farli collaborare in transazioni distribuite (JTS e JTA), per scambiarsi messaggi in modalità asincrona (JMS), per gestire l'autenticazione e le autorizzazioni (JAAS), per connettersi in modo transazionale e standard ad applicazioni esterne e/o legacy (JCA). La piattaforma J2EE nasce inoltre fortemente orientata allo sviluppo di applicazioni Internet ed Intranet con l'introduzione di un set di tecnologie (JSP e Servlet) per l'integrazione del front-end Internet/Intranet con il back-end applicativo a componenti (EJB).

1.2.2 Overview della piattaforma J2EE

La piattaforma J2EE è disegnata per fornire il supporto server-side e client-side per lo sviluppo di applicazioni enterprise multi-tier. Sebbene un'applicazione J2EE preveda tipicamente un Client Tier a fornire l'interfaccia utente, uno o più moduli Middle Tier che forniscono i servizi al client e la business logic per un'applicazione, e gli EIS di back-end a fornire data management, e può perciò consistere di svariati tier, si considerano generalmente tre tier dato che tale applicazione è distribuita su tre differenti location: macchine client, la macchina server J2EE e le macchine database o legacy al back-end. Applicazioni 3-tier siffatte estendono il classico modello client/server ponendo un Application Server multithreaded tra il client ed il back-end. La logica applicativa è divisa in componenti secondo la funzione svolta, e tali componenti sono installati su differenti macchine in base al tier cui appartengono.

- ❑ Componenti del Client Tier girano sulla macchina client.
- ❑ Componenti del Web Tier e del Business Tier (o EJB Tier) girano sul server J2EE.
- ❑ Il software dell'Enterprise Information System (EIS) Tier gira su EIS server.

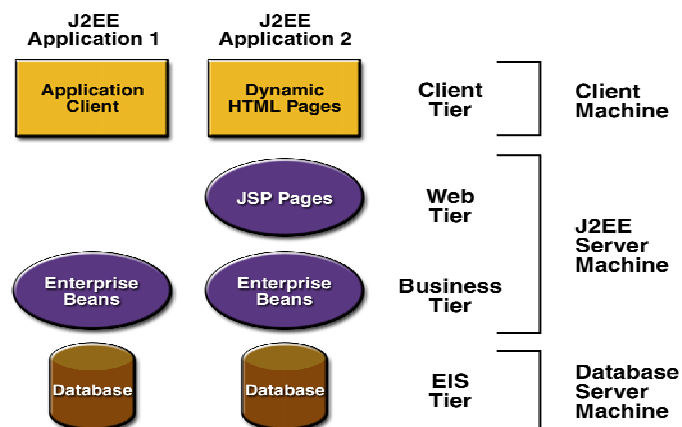


Figura 1.3 Esempio di applicazioni multi-tier

1.2.2.1 Il modello multi-tier ed il J2EE Application Model

J2EE è progettato per supportare applicazioni che implementino servizi enterprise per clienti, impiegati, fornitori, partner e chiunque altro lo richieda o interagisca con l'attività aziendale. Tali applicazioni sono implicitamente complesse, dovendo potenzialmente accedere a dati provenienti da una varietà di fonti e dovendo essere distribuite ad una varietà di client.

Per meglio controllare e gestire queste applicazioni, le business function che supportano questi diversi utenti sono gestite nel Middle Tier. Il Middle Tier rappresenta un ambiente che è attentamente controllato dal settore IT di un'azienda, che gira tipicamente su server dedicati e che ha accesso a tutti i servizi messi a disposizione dall'infrastruttura tecnologica dell'azienda. Le applicazioni J2EE spesso contano sull'EIS Tier per memorizzare i dati. Questi dati ed i sistemi che li gestiscono sono il cuore dell'azienda.

Originariamente, le soluzioni basate su applicazioni 2-tier client/server promettevano di migliorare in termini di scalabilità, efficienza e funzionalità. Tali soluzioni hanno però limiti intrinseci: la complessità della distribuzione dei servizi EIS direttamente a tutti gli utenti ed i

problemi amministrativi causati dal dover installare e mantenere la business logic su tutte le macchine utente.

Queste limitazioni legate al modello 2-tier sono superate implementando i servizi enterprise come applicazioni multi-tier. J2EE Application Model divide il lavoro necessario ad implementare un servizio multi-tier in due parti: business e presentation logic implementate dallo sviluppatore, e servizi di sistema standard forniti direttamente dalla piattaforma J2EE. Tale modello fornisce alle applicazioni multi-tier i vantaggi della portabilità WORA e della scalabilità. Questo modello standard minimizza poi i costi di formazione fornendo ampia scelta fra server e tool di sviluppo.

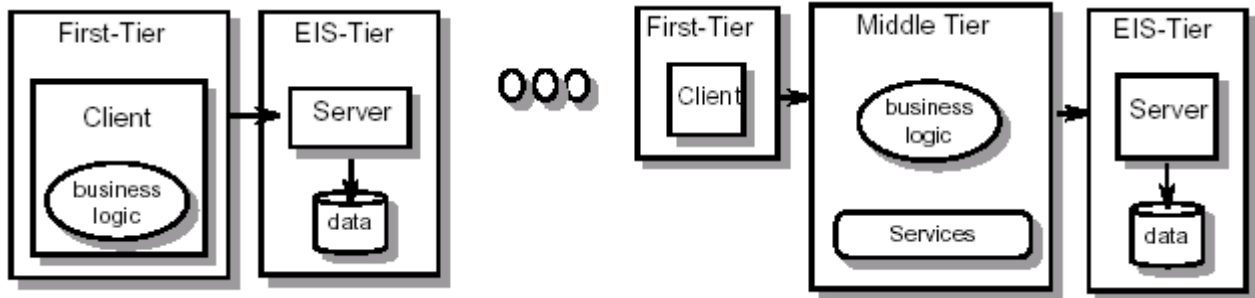


Figura 1.4 Modelli a confronto: Two-Tier vs. Multi-Tier Application

Il maggior vantaggio del modello applicativo J2EE risiede, come detto, nei Middle Tier delle applicazioni multi-tier. In tale piattaforma le middle-tier business function sono implementate come componenti EJB (o enterprise bean), come mostrato in Figura 1.5. Questi enterprise bean permettono agli sviluppatori di concentrarsi sulla business logic e lasciano all'EJB server la gestione della complessità della consegna di un servizio affidabile e scalabile.

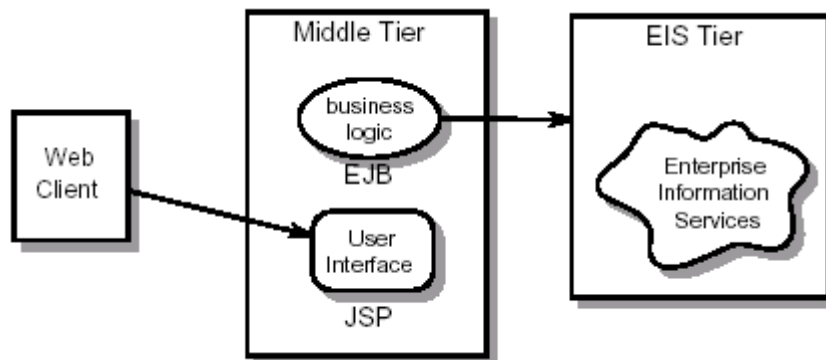


Figura 1.5 I componenti EJB implementano la business logic nel Middle Tier

La tecnologia JavaServer Pages e i servlet presentano al client i servizi implementati nel Middle Tier come servizi Internet-style semplici da accedere. La tecnologia JSP semplifica lo sviluppo e la presentazione di pagine generate dinamicamente in ciascun browser. I servlet offrono agli sviluppatori di applicazioni basate su tecnologia Java più sofisticati la libertà di implementare presentazioni dinamiche completamente in linguaggio Java.

La Figura 1.6 nella pagina seguente illustra i vari componenti e servizi che caratterizzano un tipico ambiente J2EE. Come illustrato, la piattaforma J2EE fornisce un modello per applicazioni distribuite multi-tier. Ciò significa che le varie parti di un'applicazione possono girare su dispositivi differenti. La **stratificazione orizzontale** di un'applicazione J2EE identifica un Client Tier, un Middle Tier (consistente di uno o più subtier), ed un backend tier (EIS Tier) che fornisce i servizi messi a disposizione dagli ISs esistenti. Il Client Tier supporta una molteplicità di tipi di client, residenti sia all'esterno sia all'interno dei firewall aziendali. Il Middle Tier supporta servizi per i client attraverso Web container nel Web Tier e servizi per i componenti della business logic attraverso Enterprise JavaBeans (EJB) container nell'EJB Tier. Infine, l'Enterprise Information System (EIS) Tier supporta l'accesso agli ISs esistenti per mezzo di APIs standard.

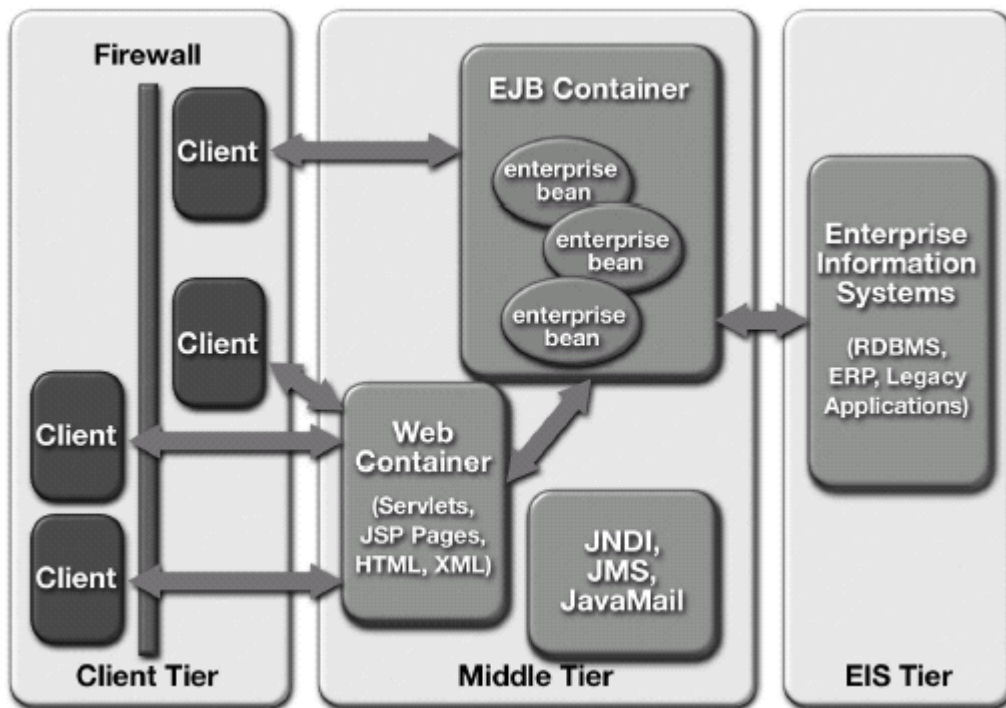


Figura 1.6 Componenti e servizi dell'ambiente J2EE

1.2.2.2 Container-Based Component Management

Nozione centrale del modello di sviluppo component-based di J2EE è quella di **container**. I container sono ambienti runtime standardizzati che forniscono specifici servizi ai componenti e che sono disponibili su tutte le piattaforme J2EE di ciascun vendor. Per esempio, tutti i J2EE Web container forniscono supporto runtime per rispondere a richieste client, processare tali richieste (invocando per esempio JSP o servlet), e ritornare i risultati al client. Tutti gli EJB container forniscono supporto automatico alla gestione delle transazioni e del ciclo di vita di un componente EJB, così come alla ricerca del componente (bean lookup) e altri servizi. I container forniscono anche accessi standardizzati agli EISs: per esempio l'accesso ad un RDBMS è reso possibile dall'API JDBC.

Inoltre, i container forniscono un meccanismo per impostare l'application behavior al momento dell'assemblaggio o del deployment, attraverso l'uso dei **deployment descriptor** (file di testo che specificano il behavior del componente in termini di tag XML ben definiti). Caratteristiche come controllo della sicurezza e controllo delle transazioni possono essere configurate al momento del deployment (senza dover essere gestite a livello di codice del componente).

Le J2EE Specification definiscono i container che devono essere supportati, ma non specificano o restringono la configurazione di questi container. Così, sia Web container sia EJB container potrebbero girare sulla stessa piattaforma, oppure i Web container potrebbero vivere su una piattaforma e gli EJB container su un'altra, od infine una piattaforma J2EE potrebbe essere composta da molteplici container su molteplici piattaforme.

1.2.2.3 Supporto per componenti client

Il J2EE Client Tier fornisce supporto per una molteplicità di tipi di client, sia all'interno sia all'esterno del firewall aziendale. I client si possono presentare come browser Web usando pagine HTML statiche, HTML dinamico generato con la tecnologia JavaServer Pages, o applet Java; oppure possono presentarsi come applicazioni Java stand-alone. Si suppone che i client J2EE accedano al Middle Tier principalmente usando gli standard Web, cioè HTTP, HTML e XML.

Per supportare interazioni più complesse con l'utente, può essere necessario fornire funzionalità direttamente nel Client Tier. Questa caratteristica è tipicamente implementata attraverso i componenti JavaBeans che interagiscono con il servizio nel Middle Tier via servlet. I componenti JavaBeans (o **bean**) client-tier saranno tipicamente forniti dal servizio come un applet

scaricata automaticamente nel browser dell'utente. Per eliminare i problemi causati da versioni della JVM vecchie e non-standard nel browser dell'utente, il modello di applicazione J2EE fornisce uno speciale supporto per scaricare ed installare automaticamente il Java Plug-in.

1.2.2.4 Supporto per componenti Business Logic

Nella piattaforma J2EE, la business logic è implementata nel Middle Tier come componenti EJB (o **enterprise bean**). La tecnologia EJB permette allo sviluppatore di concentrarsi sulla business logic in quanto la complessità della consegna di un servizio affidabile e scalabile è gestita dall'EJB server.

La piattaforma J2EE e l'architettura EJB hanno scopi complementari. L'EJB component model è il backbone del modello di programmazione J2EE. La piattaforma J2EE complementa le specifiche EJB specificando le APIs che un EJB developer può utilizzare per implementare gli enterprise bean e definendo il più ampio e distribuito ambiente di programmazione nel quale gli EJB sono usati come componenti business logic.

1.2.2.5 Supporto allo standard J2EE

Lo standard J2EE, come visto, è definito attraverso un insieme di specifiche collegate. Insieme, queste specifiche descrivono l'architettura. In aggiunta alle specifiche, diverse altre proposte sono disponibili per supportare lo standard J2EE, inclusi la J2EE Compatibility Test Suite (CTS) e il J2EE Software Development Kit (SDK).

La CTS aiuta a massimizzare la portabilità delle applicazioni convalidando la compatibilità di un prodotto J2EE con la specifica standard, e partendo da dove il Java Conformance Kit (JCK) ha lasciato. La CTS testa la conformità alle API estensione di Java standard non previste dal JCK. In aggiunta, testa la capacità di una piattaforma J2EE di far girare applicazioni standard.

Il J2EE SDK è l'implementazione di riferimento di J2EE ed è destinato a raggiungere diversi obiettivi. Primo, fornisce una definizione operativa della piattaforma J2EE, usata dai vendor come "goal standard" per determinare ciò che i loro prodotti devono fare sotto un particolare insieme di ipotesi applicative. Può essere usato dagli sviluppatori per verificare la portabilità di una applicazione. Ed è usato come piattaforma standard per far girare la CTS.

Secondo, con il J2EE SDK Sun rende disponibile gratuitamente alla comunità di sviluppatori un'implementazione della piattaforma J2EE per aiutarli ad adottare velocemente lo standard J2EE. Sebbene non sia un prodotto commerciale ed i termini della licenza ne proibiscano il suo uso commerciale, il J2EE SDK è disponibile gratuitamente e può essere usato nello sviluppo di demo applicative e prototipi.

1.2.3 I vantaggi della piattaforma J2EE

Con un set di caratteristiche appositamente progettate per accelerare il processo di sviluppo di applicazioni distribuite, la piattaforma J2EE offre diversi vantaggi:

- Sviluppo ed architettura semplificati
- Facile scalabilità per venire incontro a variazioni di domanda (ad aumenti di richieste)
- Integrazione con i ISs esistenti
- Ampia scelta di server, tool e componenti
- Modello di sicurezza flessibile

1.2.3.1 Sviluppo ed architettura semplificati

La piattaforma J2EE supporta un modello di sviluppo basato su componenti semplificati. Essendo basata sul linguaggio di programmazione Java e sulla Java™ 2 Platform, Standard Edition (**J2SE™**), questo modello offre la portabilità sintetizzata dal paradigma Write Once, Run Anywhere™: un'applicazione risulta portabile su tutti i server conformi allo standard J2EE.

Tale modello può aumentare la produttività nello sviluppo di applicazioni in diversi modi:

- ✚ *Si adatta facilmente e flessibilmente alle funzionalità richieste per un'applicazione:* i modelli di applicazioni component-based rappresentano facilmente e flessibilmente le funzionalità

desiderate da un'applicazione. J2EE fornisce svariate possibilità di configurare l'architettura di un'applicazione, dipendentemente dal tipo di client richiesto, dal livello di accesso alle fonti dati richiesto, ecc... Il modello component-based inoltre semplifica la manutenzione dell'applicazione, dato che i componenti possono essere aggiornati e sostituiti in maniera indipendente. Ad esempio, nuove funzionalità possono essere aggiunte ad applicazioni esistenti semplicemente aggiornando i componenti selezionati.

✚ *Permette di specificare il comportamento (behavior) dell'applicazione in fase di assemblaggio e deployment:* i componenti possono contare sulla disponibilità di servizi standard forniti dall'ambiente runtime, e possono essere connessi dinamicamente ad altri componenti tramite interfacce well-defined. Da ciò, i comportamenti di molte applicazioni possono essere configurati in fase di assemblaggio e deployment, senza alcuna ricodifica richiesta. Gli sviluppatori di componenti possono comunicare i loro requisiti agli Application Deployer attraverso specifiche impostazioni. Esistono tool per automatizzare questo processo al fine di facilitare ed accelerare lo sviluppo.

✚ *Supporta la divisione del lavoro:* i componenti aiutano a dividere il lavoro di sviluppo tra specifici skill set, permettendo ad ogni membro del team di sviluppo di focalizzarsi sulle proprie capacità e competenze. Così, i template JSP possono essere creati dai designer grafici, i loro behavior da programmatori Java, la business logic da domain expert, e l'assemblaggio ed il deployment dell'applicazione dagli appropriati membri del team. Questa divisione del lavoro aiuta anche ad accelerare la manutenzione dell'applicazione. Si consideri l'interfaccia utente, che è la parte più dinamica di molte applicazioni, particolarmente sul Web. Con la piattaforma J2EE, i designer grafici possono modificare il look dell'interfaccia utente JSP-based senza la necessità di far intervenire alcun programmatore. Le specifiche descrivono diversi ruoli generici, tra i quali Application Component Provider, Application Assembler e Application Deployer. In alcuni team di sviluppo, uno o due persone potrebbero interpretare tutti questi ruoli.

1.2.3.2 Facile scalabilità

I container della piattaforma J2EE forniscono un meccanismo che permette di ottenere un alto livello di scalabilità delle applicazioni distribuite semplicemente, senza richiedere alcuno sforzo ed alcun intervento da parte del team di sviluppo dell'applicazione.

Dato che i container forniscono ai componenti il supporto alle transazioni, alle connessioni a database, alla gestione del ciclo di vita e ad altre caratteristiche che influenzano la performance, essi possono essere progettati affinché realizzino la scalabilità rispetto a ciascuna di queste funzionalità. Per esempio, fornendo il pooling delle connessioni a database, i contenitori possono assicurare che i client abbiano un rapido accesso ai dati.

Inoltre, dato che le specifiche lasciano ai server provider la libertà di configurare i container per essere eseguiti su più sistemi, i container possono essere implementati per realizzare sistemi di bilanciamento automatico del carico

1.2.3.3 Integrazione con gli EISs esistenti

La piattaforma J2EE, insieme alla J2SE, include una varietà di APIs standard per accedere agli EISs esistenti ed aggiornare le informazioni in essi contenute. L'accesso di base a questi sistemi è fornito dalle seguenti APIs:

- **JDBC™** (Java DataBase Connectivity) è l'API per accedere a database relazionali da Java.
- **JTA™** (Java Transaction API) è l'API per gestire e coordinare transazioni distribuite attraverso EISs eterogenei.
- **JNDI™** (Java Naming and Directory Interface) è l'API per accedere alle informazioni dei servizi di naming e directory dell'azienda.
- **JMS™** (Java Message Service) è l'API per inviare e ricevere messaggi attraverso sistemi di messaging aziendali come IBM MQ Series e TIBCO Rendez-vous.
- **JavaMail™** è l'API per inviare e ricevere e-mail.
- **Java IDL** è l'API per invocare servizi CORBA.

In aggiunta, accessi specializzati a sistemi Enterprise Resource Planning e sistemi mainframe aziendali, come CICS di IBM e IMS, saranno forniti in versioni future delle specifiche attraverso J2EE Connector Architecture.

1.2.3.4 Ampia scelta di server, tool e componenti

Lo standard ed il brand J2EE hanno lo scopo di creare un vivace ed ampio mercato per server, tool e componenti. Il marchio J2EE su un prodotto server assicura quella specie di ubiquità che è fondamentale per gli scopi della piattaforma J2EE. Lo standard J2EE assicura un vivace mercato anche per tool e componenti.

- ✚ *Ampia scelta di server:* le aziende possono contare su una moltitudine di piattaforme compatibili con J2EE da vari produttori, i quali forniscono un'ampia scelta tra server, che si differenziano per piattaforme hardware, sistemi operativi e configurazioni. Questo assicura alle aziende di poter operare la scelta più appropriata agli scopi strategici delle applicazioni di cui necessitano.
- ✚ *Scelta di tool grafici per lo sviluppo, il deployment e la gestione delle applicazioni:* sia i componenti EJB sia le pagine JSP sono disegnati per essere manipolati da tool grafici di sviluppo, e per permettere di automatizzare molti dei processi di sviluppo che richiedono tradizionalmente la capacità di scrivere e debuggare il codice. Sia i J2EE server provider sia i third-party tool developer possono sviluppare tool che si adattano agli standard J2EE e supportano vari processi e stili di sviluppo. Gli sviluppatori possono così scegliere fra una varietà di tool per manipolare ed assemblare componenti.
- ✚ *Un mercato per i componenti:* la progettazione basata su componenti assicura di poter standardizzare, impacchettare (package) e riutilizzare molti tipi di behavior e funzionalità. I vendor di componenti forniranno una varietà di componenti off-the-shelf, inclusi accounting bean, template per interfacce utente e componenti specifiche per funzionalità particolari utili in particolari settori di business. Gli sviluppatori hanno così a disposizione una scelta di componenti standardizzate.

Quest'ampia libertà di scelta permessa da J2EE rende possibile sviluppare applicazioni con la certezza e la fiducia che il valore dell'investimento sarà protetto.

1.2.3.5 Modello di sicurezza semplificato ed unificato

Il modello di sicurezza di J2EE è disegnato per supportare accessi signon singoli ai servizi applicativi. Gli sviluppatori di componenti possono specificare i requisiti di sicurezza di un componente a livello di metodo, per assicurare che solo utenti con permessi appropriati possano accedere a specifiche operazioni sui dati. Anche se entrambe le APIs EJB e Java Servlet forniscono un meccanismo per costruire controlli di sicurezza all'interno del codice, il meccanismo di base per fare il matching tra utenti e ruoli (gruppi di utenti hanno permessi specifici) è realizzato interamente al momento del deployment dell'applicazione. Questo fornisce sia maggior flessibilità sia miglior controllo sulla sicurezza.

Mentre altri modelli richiedono misure di sicurezza platform-specific, l'ambiente di sicurezza della piattaforma J2EE permette, come detto, di definire i vincoli di sicurezza al momento del deployment. Proteggendo e nascondendo alle applicazioni la complessità dell'implementazione della sicurezza, la piattaforma J2EE rende le applicazioni portabili ad un largo insieme di security implementation.

Essa definisce regole di controllo dell'accesso dichiarative standard che devono essere definite dal programmatore/assemblatore dell'applicazione e interpretate quando questa è deployata sulla piattaforma aziendale. Inoltre J2EE richiede ai vendor di fornire meccanismi standard di login così che le applicazioni non devono incorporare tali meccanismi all'interno della loro logica. Lo stesso programma lavora in una varietà di ambienti di sicurezza differenti senza alcun cambiamento al codice sorgente.

Per esempio, uno sviluppatore J2EE può specificare diversi livelli di sicurezza (user, super-user, administrator, ecc...) , poi scrivere il codice per controllare il livello di permesso dell'utente corrente per accedere ad operazioni sicure. Al momento del deployment, l'Application Deployer

asigna gruppi di utenti agli appropriati livelli di sicurezza, permettendo così all'applicazione di verificare facilmente il livello di permesso prima di realizzare operazioni il cui accesso è ristretto.

La sicurezza è quindi quasi completamente gestita dalla piattaforma e dai suoi amministratori. Nella maggior parte dei casi, né il servizio né il suo client richiedono allo sviluppatore la scrittura di security logic.

1.3 Scenari possibili per Applicazioni J2EE

Le sezioni seguenti presentano diversi scenari applicativi e mettono in evidenza come la piattaforma J2EE sia **funzionalmente completa**. La Figura 1.7 riflette alcuni scenari chiave che un prodotto J2EE dovrebbe essere capace di supportare, inclusi quelli nei quali o il Web container o l'EJB container od entrambi sono bypassati.

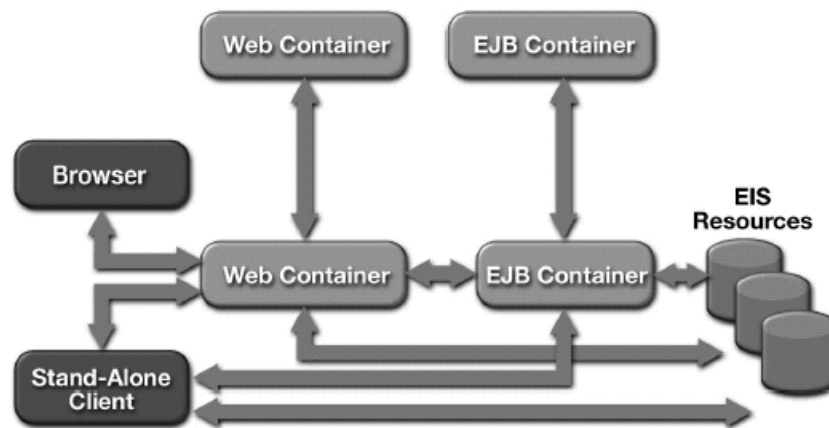


Figura 1.7 Scenari possibili per applicazioni J2EE

Quelli di seguito elencati sono classici requisiti per un'applicazione multi-tier. Si assume la presenza sia del Web container sia dell'EJB container.

- ◆ Necessità di fare rapidi e frequenti cambiamenti al look dell'applicazione.
- ◆ Necessità di partizionare l'applicazione tra logica di presentazione e logica di business così da incrementare la modularità.
- ◆ Necessità di semplificare il processo di assegnamento di risorse umane adeguatamente formate per compiere il processo di sviluppo in modo tale che il lavoro proceda su binari indipendenti ma cooperanti.
- ◆ Necessità di avere sviluppatori familiari con applicazioni back-office sollevati dal lavoro di design grafico e GUI, per il quale essi non possono essere idealmente qualificati.
- ◆ Necessità di avere il vocabolario necessario per comunicare la business logic a team legati ai fattori umani ed all'estetica dell'applicazione.
- ◆ Capacità di assemblare applicazioni back-office usando componenti da una varietà di fonti, inclusi componenti off-the-shelf business logic
- ◆ Abilità di deployare componenti transazionali tra piattaforme hardware e software diverse indipendentemente dalla tecnologia database sottostante.
- ◆ Abilità di proiettare esternamente dati interni senza dover fare molte assunzioni sull'utente che elaborerà dati.

Chiaramente rilassare qualcuno o tutti questi requisiti influenzerebbe alcune delle decisioni e scelte che un progettista deve prendere a livello applicativo.

J2EE promuove un modello di programmazione che anticipa la crescita, incoraggia la riusabilità del codice component-oriented, e fa leva sulla forza della comunicazione inter-tier. Il cuore dello standard J2EE è rappresentato appunto dall'integrazione tra i vari tier.

1.3.1 Scenario di Applicazione Multi-tier

La Figura 1.8 illustra uno scenario applicativo nel quale il Web container ospita componenti Web che sono quasi esclusivamente dedicati alla gestione della logica di presentazione dei dati. La consegna di contenuti Web dinamici al client è responsabilità delle pagine JSP (supportate da servlet). L'EJB container ospita i componenti che, da un lato, rispondono alle richieste dal Web Tier, e dall'altro, accedono alle risorse degli EISs. La capacità di disaccoppiare l'accesso ai dati dalle interazioni con l'utente finale è la forza di questo particolare scenario. L'applicazione è implicitamente scalabile, ma decisamente più importante è il fatto che le funzionalità applicative di back-office sono relativamente isolate e separate dalla presentazione per l'utente finale.

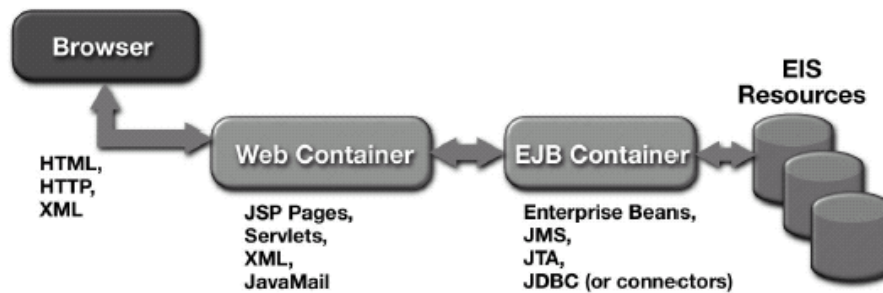


Figura 1.8 Applicazione multi-tier

Considerando il Web Tier, una questione ricorrente è se usare pagine JSP o servlet. Il modello di programmazione J2EE promuove la tecnologia JSP quale strumento di programmazione all'interno del Web container. Le pagine JSP contano sulle funzionalità messe a disposizione dai servlet ma il modello evidenzia il fatto che le pagine JSP si adattano più facilmente ai Web engineer. Dunque il Web container è ottimizzato per la creazione di contenuti dinamici destinati ai client Web usando la tecnologia JSP; l'uso di servlet dovrebbe essere visto piuttosto come un'eccezione.

1.3.2 Scenario di Client Stand-alone

La Figura 1.9 illustra uno scenario di client stand-alone.

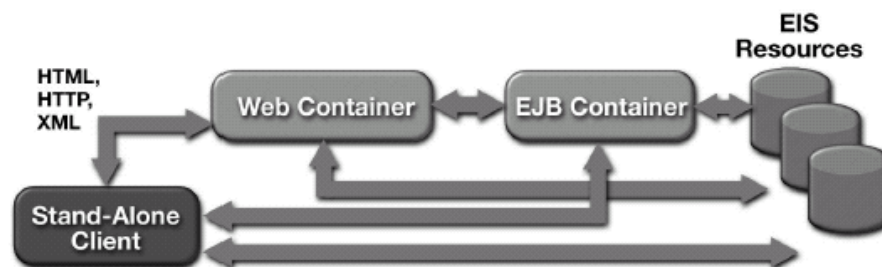


Figura 1.9 Client stand-alone

Si devono considerare tre tipi di client stand-alone:

- ✚ **EJB client** che interagiscono direttamente con un EJB server, cioè con un componente EJB ospitato da un EJB container. Tale scenario è illustrato nella Figura 1.10 alla pagina seguente. Si assume che sia usato RMI-IIOP e che l'EJB server acceda alle risorse degli EISs usando JDBC (connettori in futuro).

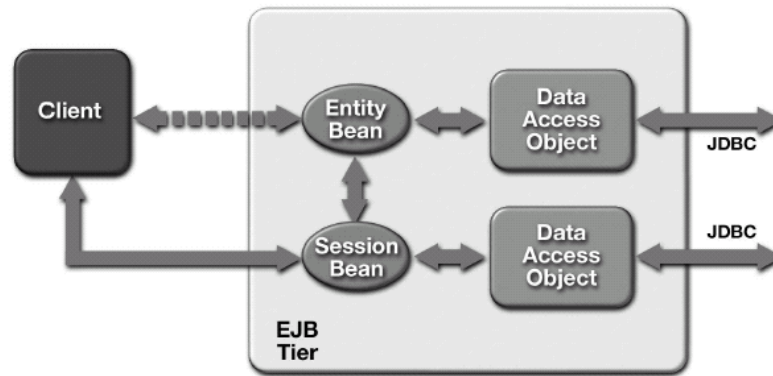


Figura 1.10 Client Java EJB-centric

- ✦ **Applicazioni client Java stand-alone** che accedono alle risorse degli EISs direttamente usando JDBC (e in futuro anche connettori). In tale scenario la presentation logic e la business logic sono per definizione co-located sulla piattaforma client e possono di fatto essere strettamente connesse all'interno di una singola applicazione. Il Middle Tier è collassato dentro la piattaforma client, e lo scenario rappresenta nient'altro che una classica applicazione client/server con i relativi problemi di distribuzione, manutenzione e scalabilità.
- ✦ **Client Visual Basic** che consumano contenuti Web dinamici, per lo più nella forma di XML data message. In questo scenario, il Web container gestisce fundamentalmente le trasformazioni XML e fornisce connettività Web ai client. Si suppone che la presentation logic sia gestita sul Client Tier. Il Web Tier può essere progettato per gestire la business logic e per accedere direttamente alle risorse degli EISs.

1.3.3 Scenario di Applicazione Web-centric

La Figura 1.11 illustra uno scenario di applicazione 3-tier Web-centric.

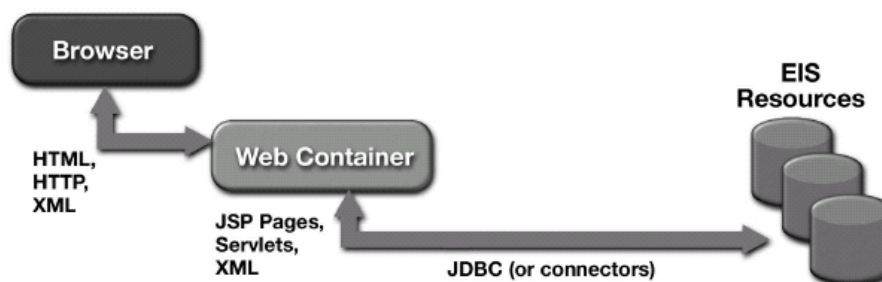


Figura 1.11 Applicazione 3-tier Web-centric

Tale scenario (3-tier) è molto diffuso. Fundamentalmente il Web container ospita sia la presentation sia la business logic, e si suppone che siano usati JDBC o, in un prossimo futuro, connettori, per accedere alle risorse degli EISs.

La successiva Figura 1.12 fornisce una visione più dettagliata del Web container in un tale scenario.

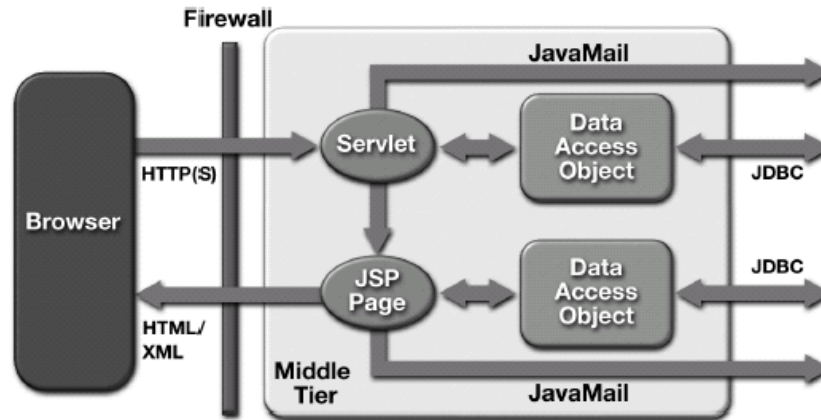


Figura 1.12 Web container in uno scenario 3-Tier

1.3.4 Scenario di B2B

La Figura 1.13 illustra uno scenario B2B.

Questo scenario focalizza l'attenzione sulle interazioni peer-level tra Web container ed EJB container. Il modello di programmazione J2EE promuove l'uso di XML data messaging over HTTP come principale modalità per stabilire comunicazioni tra gli EJB container. Questa caratteristica si adatta naturalmente allo sviluppo ed al deployment di soluzioni per il commercio elettronico Web-based.

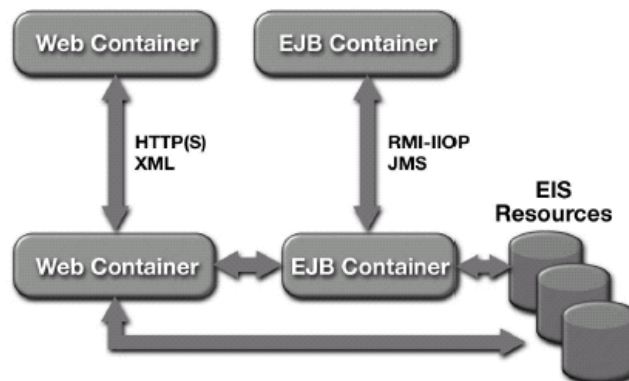


Figura 1.13 Scenario di B2B

1.4 Packaging e Deployment di una Applicazione J2EE

Una applicazione J2EE è impacchettata (packaged) in una o più unità standard per il deployment su ciascun sistema J2EE-compatibile.

Ogni unità contiene uno o più componenti funzionali (enterprise bean, pagina JSP, servlet, applet, ecc.), un deployment descriptor standard che descrive il suo contenuto, e le J2EE declaration (dichiarazioni XML-based specificate dall'Application Developer o dall'Application Assembler e memorizzate nel deployment descriptor, le quali permettono all'Application Deployer di modificare il comportamento di una applicazione senza dover modificare i componenti stessi). Una volta che un'unità J2EE è stata prodotta, essa è pronta per essere deployata, come mostrato in Figura 1.14.

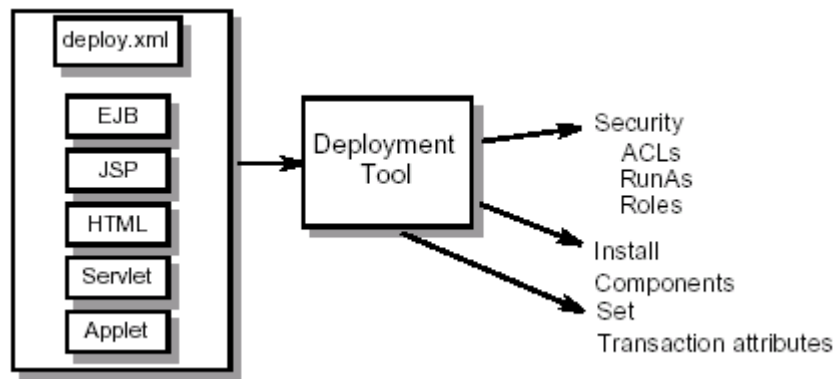


Figura 1.14 Packaging e deployment di una applicazione J2EE

Il deployment tipicamente implica l'uso di un tool per specificare informazioni location-specific. Una volta deployata sulla piattaforma locale, l'applicazione è pronta per essere eseguita.

La piattaforma J2EE permette agli sviluppatori di creare parti diverse delle loro applicazioni come componenti riusabili. Il processo di assembling dei componenti all'interno di moduli e di moduli all'interno delle applicazioni enterprise è chiamato **packaging**. In un buon disegno software, i componenti riusabili possono essere customizzati per l'ambiente in cui opereranno. Il processo di installazione e customizzazione di un'applicazione nell'ambiente in cui opererà è chiamato **deployment**. Per permettere la customizzazione, i componenti di un'applicazione devono essere configurabili. E' necessario un meccanismo che fornisca flessibilità nella configurazione e supporti strumenti che aiutino tale processo.

La piattaforma J2EE fornisce delle modalità per rendere semplici i processi di packaging e deployment. Infatti prevede l'uso di file JAR come package standard per moduli ed applicazioni, e deployment descriptor XML-based per la customizzazione dei componenti e delle applicazioni.

1.4.1 Ruoli e Processi

Il packaging ed il deployment coinvolgono tre diversi ruoli: Application Component Provider, Application Assembler, e Deployer.

L'**Application Component Provider** (ACP) sviluppa enterprise bean, pagine HTML e JSP. Gli ACP forniscono le informazioni strutturali del deployment descriptor di ciascun componente. Tali informazioni includono le interfacce home e remote e le classi che implementano gli enterprise bean, i meccanismi di persistenza usati, e il tipo di risorse che i componenti usano, informazioni queste facenti parte del codice dell'applicazione e non configurabili al momento del deployment. Il codice di esempio contiene un estratto del deployment descriptor di un ipotetico Entity Bean:

```
<entity>
  <display-name>TheAccount</display-name>
  <ejb-name>TheAccount</ejb-name>
  <home>com.sun.estore.account.ejb.AccountHome</home>
  <remote>com.sun.estore.account.ejb.Account</remote>
  <ejb-class>com.sun.estore.account.ejb.AccountEJB</ejb-class>
  <persistence-type>Bean</persistence-type>
  <prim-key-class>java.lang.String</prim-key-class>
  <reentrant>False</reentrant>
  <resource-ref>
    <description>description</description>
    <res-ref-name>jdbc/EstoreDataSource</res-ref-name>
    <res-type>javax.sql.DataSource</res-type>
    <res-auth>Container</res-auth>
  </resource-ref>
</entity>
```

Esempio 1.1 Deployment Descriptor per un Entity Bean

L'**Application Assembler** (AA) fornisce informazioni riguardanti l'intera applicazione. Per esempio, l'AA potrebbe configurare il file `Main.jsp` in maniera tale che gestisca richieste provenienti dall'URL namespace (`/control/*`), le pagine di errore usate dall'applicazione, i suoi vincoli di sicurezza e ruoli, e così via. Il codice di esempio contiene un estratto del deployment descriptor di una applicazione Web:

```
<web-app>
  <display-name>JavaPetStoreDemoWebTier</display-name>
  <servlet>
    <servlet-name>webTierEntryPoint</servlet-name>
    <display-name>centralJsp</display-name>
    <description>central point of entry for the Web app</description>
    <jsp-file>Main.jsp</jsp-file>
  </servlet>
  <servlet-mapping>
    <servlet-name>webTierEntryPoint</servlet-name>
    <url-pattern>/control/*</url-pattern>
  </servlet-mapping>
  ...
  <error-page>
    <exception-type>java.lang.Exception</exception-type>
    <location>/errorpage.jsp</location>
  </error-page>
  ...
</web-app>
```

Esempio 1.2 Deployment Descriptor per una Applicazione Web

Un **Deployer** è responsabile del deployment dei componenti e delle applicazioni all'interno dell'ambiente operativo. Il deployment tipicamente coinvolge due processi:

1. **Installazione** – Il Deployer sposta i file multimediali sul server, genera le classi e le interfacce container-specific aggiuntive che permettono al container la gestione runtime dei componenti, e installa i componenti e le classi ed interfacce sul server J2EE.
2. **Configurazione** - Il Deployer risolve tutte le dipendenze esterne dichiarate dall'ACP e segue le istruzioni per l'assembly dell'applicazione definite dall'AA. Per esempio, il Deployer è responsabile dell'associazione tra i ruoli di sicurezza definiti dall'AA ed i gruppi e gli account utente esistenti nell'ambiente operativo all'interno del quale i componenti e le applicazioni sono deployate.

La Figura 1.15 illustra i ruoli, appena descritti, coinvolti nei processi di packaging e deployment di un'applicazione J2EE.

Roles	Tasks
Application Component Provider	Specify component deployment descriptors. Package components into modules.
Application Assembler	Resolve dependencies between deployment descriptor elements in different modules. Assemble modules into larger deployment units.
Deployer	Customize deployment descriptor elements for environment. Install deployment units into server(s).

Figura 1.15 Ruoli coinvolti nei processi di Packaging e Deployment

1.4.2 Il Packaging di Applicazioni J2EE

Un'applicazione J2EE è impacchettata in un file **Enterprise Archive** (EAR), un file Java ARchive (JAR) standard con estensione **.ear**. Lo scopo di questo formato di file è di fornire un'unità di deployment portabile. Un tale file contiene uno o più moduli J2EE ed un deployment descriptor. Dunque, la creazione di un'applicazione J2EE è un processo two-step. Primo, l'ACP crea i moduli EJB, Web e client. Secondo, l'AA impacchetta questi moduli insieme a creare l'applicazione J2EE.

1.4.3 I Deployment Descriptor

Un deployment descriptor è un file di testo XML-based i cui elementi descrivono come assemblare e deployare l'unità, cui si riferisce il file, all'interno di uno specifico ambiente. Ciascun elemento consiste di un tag ed un valore espressi nella sintassi `<tag>value</tag>`.

Solitamente i deployment descriptor sono generati automaticamente da appositi tool per il deployment, così da non doverli gestire direttamente. Gli elementi contengono informazioni sul behavior dei componenti non incluse direttamente nel codice. Il loro scopo è di informare il Deployer su come deployare un'applicazione, e non di dire al server come gestire runtime i componenti. Esistono diversi tipi di deployment descriptor: EJB deployment descriptor, Web deployment descriptor, Application deployment descriptor ed Application client deployment descriptor. Essi specificano due tipi di informazioni:

- ✦ Informazioni strutturali, che descrivono i diversi componenti del file JAR, le loro relazioni e le loro dipendenze esterne.
- ✦ Informazioni sull'assemblaggio, che descrivono come i contenuti del file JAR possono essere composti in un'unità deployabile. Sono opzionali.

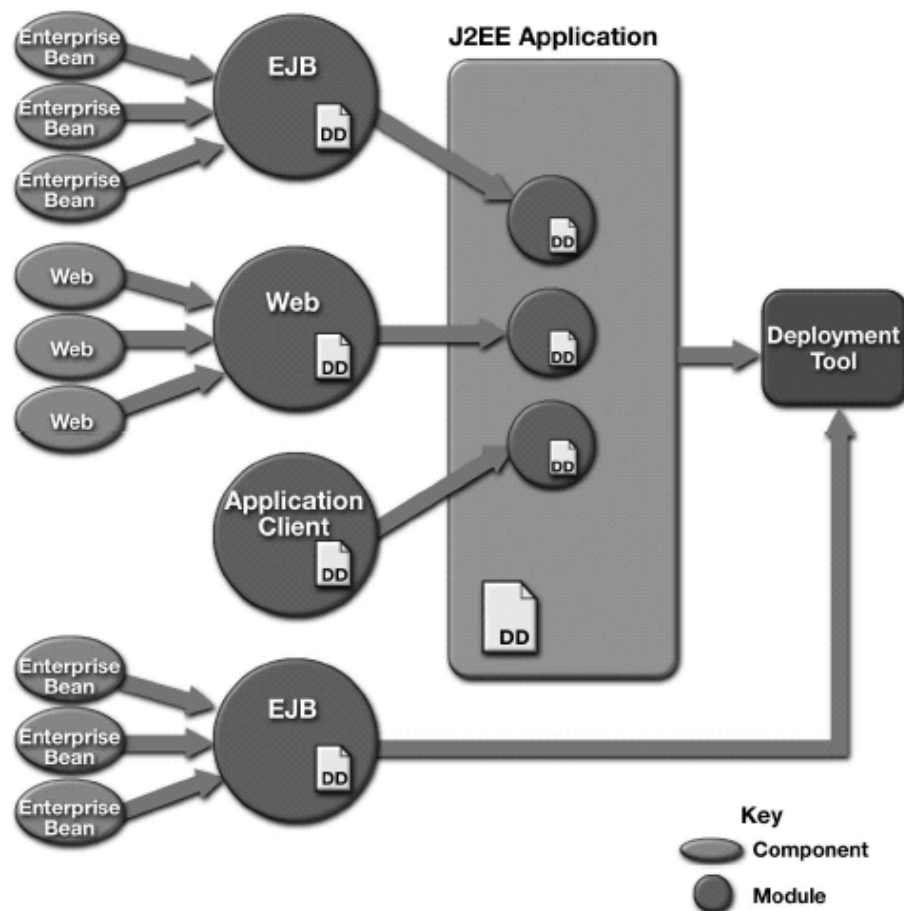


Figura 1.16 Package J2EE

1.5 Riassunto

Oggi la sfida lanciata ai professionisti dell'IT è di sviluppare e deployare efficientemente applicazioni distribuite da usare sia sulle Intranet aziendali sia su Internet. Le compagnie che riusciranno a fare ciò efficacemente guadagneranno vantaggi strategici nell'Information Economy.

La piattaforma J2EE è un insieme standard di tecnologie Java che semplificano e rendono più efficiente lo sviluppo, il deployment e la gestione delle applicazioni enterprise. La piattaforma J2EE è funzionalmente completa nel senso che è possibile sviluppare una larga classe di applicazioni enterprise usando solo le tecnologie J2EE. Le applicazioni scritte per la piattaforma J2EE gireranno su ciascun server J2EE-compatibile.

La piattaforma fornisce svariati vantaggi alle organizzazioni per quanto riguarda lo sviluppo di tali applicazioni, inclusi un modello di sviluppo semplificato, facile scalabilità, supporto agli EISs esistenti, scelte tra server, tool e componenti ed un modello semplice e flessibile di sicurezza.

Capitolo 2 Le tecnologie della piattaforma J2EE

La piattaforma J2EE specifica un insieme di tecnologie per supportare applicazioni enterprise. Queste tecnologie si suddividono in tre categorie: component, service e communication.

I componenti sono le tecnologie usate dagli sviluppatori per creare le parti essenziali di un'applicazione enterprise, cioè l'interfaccia utente e la business logic. Tali tecnologie permettono lo sviluppo di moduli che possono essere riutilizzati da varie applicazioni enterprise. Sono supportati dai servizi system-level della piattaforma J2EE che semplificano la programmazione e permettono ai componenti di essere customizzati per usare le risorse disponibili nell'ambiente nel quale sono deployati.

Dato che la maggior parte delle applicazioni enterprise richiedono accesso agli EISs esistenti, la piattaforma J2EE supporta APIs che forniscono servizi di accesso a database, di gestione delle transazioni, di naming e directory e di messaging. Infine, la piattaforma fornisce tecnologie che permettono la comunicazione tra client e server e tra oggetti collaboranti situati in differenti server.

2.1 Component Technologies

Cosa è un componente? Un **componente** è un oggetto caricabile e gestibile all'interno di un container ed in grado di sfruttarne i servizi messi a disposizione.

Più in dettaglio un componente è un elemento software a livello di applicazione (application-level software unit) self-contained, assemblato in un'applicazione J2EE assieme alle classi ed i file che lo riguardano, e che comunica con altri componenti.

In aggiunta ai componenti JavaBeans, che sono parte della piattaforma J2SE, la piattaforma J2EE supporta i seguenti tipi di componenti: applet, application client, componenti Enterprise JavaBeans e componenti Web. Applet e application client sono tecnologie client-side che girano quindi su una piattaforma client (grazie al supporto della JVM ivi presente), mentre i componenti EJB e Web sono tecnologie server-side e girano quindi su una piattaforma server.

Tutti i componenti J2EE dipendono dal supporto runtime fornito da un'entità system-level chiamata **container**. I container forniscono ai componenti servizi quali gestione del ciclo di vita, sicurezza, deployment e threading. Dato che i container gestiscono questi servizi, i behavior di molti componenti possono essere definiti e customizzati quando il componente è deployato nel container.

2.1.1 Applet e Application Client

Una pagina Web ricevuta dal Web Tier può includere un'applet incorporata. Un'**applet** è una piccola applicazione client scritta in Java che esegue nella JVM installata nel browser Web, e che potrebbe aver bisogno del Java Plug-in, il quale fornisce l'ambiente di esecuzione necessario per la corretta esecuzione delle applet.

Un **application client** gira su una macchina client e fornisce un modo per gli utenti di gestire processi che richiedono un'interfaccia grafica più ricca di quella fornita da un linguaggio di markup. Generalmente tali applicazioni hanno una GUI (Graphic User Interface) creata attraverso le APIs Swing o Abstract Window Toolkit (AWT), ma è ovviamente possibile anche un'interfaccia command-line.

2.1.2 Componenti Web

Un **componente Web** è un'entità software che sostanzialmente fornisce una risposta ad una richiesta. Un componente Web tipicamente genera l'interfaccia utente per un'applicazione Web-based.

La piattaforma J2EE specifica 2 tipi di componenti Web: servlet (versione attuale delle specifiche 2.3 disponibile all'indirizzo <http://java.sun.com/products/servlet>) e pagine JavaServer Pages (versione attuale delle specifiche 1.2 disponibile all'indirizzo <http://java.sun.com/products/servlet>).

2.1.2.1 La tecnologia Servlet

Un **servlet** è un programma che estende le funzionalità di un server Web. I servlet sono classi Java che ricevono una richiesta da un client, processano la richiesta, generano dinamicamente la risposta (eventualmente interrogando un database), ed infine inviano la risposta contenente un documento HTML o XML al client. Uno sviluppatore ricorre alla Java Servlet API per:

- ✦ Inizializzare e completare un servlet
- ✦ Accedere all'ambiente del servlet
- ✦ Ricevere richieste ed inviare risposte
- ✦ Mantenere informazioni di sessione
- ✦ Interagire con altri servlet o altri componenti

2.1.2.2 La tecnologia JavaServer Pages

La tecnologia **JavaServer Pages** fornisce un modo estendibile per generare contenuti dinamici per un client Web. Una pagina JSP è un documento text-based che descrive come processare una richiesta per creare una risposta. Essa contiene:

- ✦ Template Data per formattare il documento Web. Tipicamente il template data usa elementi HTML o XML. I designer grafici possono editare e lavorare con questi elementi sulle pagine JSP, senza influenzare il contenuto dinamico. Questo approccio semplifica lo sviluppo poiché separa la logica di presentazione da quella di generazione del contenuto dinamico.
- ✦ Elementi JSP e scriptlet per generare il contenuto dinamico nel documento Web. La maggior parte delle pagine JSP usano componenti JavaBeans e/o Enterprise JavaBeans per realizzare le elaborazioni più complesse richieste dall'applicazione. Le action standard di JSP possono accedere ed instanziare bean, settare o recuperare attributi, e scaricare applet. JSP è estendibile attraverso lo sviluppo di custom action, che sono incapsulate in tag library.

2.1.2.3 Il Web Component Container

I componenti Web sono ospitati su servlet container, JSP container e Web container. In aggiunta ai servizi standard di un container, un servlet container fornisce servizi di rete (grazie ai quali sono spedite richieste e risposte), decodifica le richieste, e formatta le risposte. Tutti i servlet container devono supportare HTTP come protocollo request/response, ma possono supportare anche protocolli request/response addizionali come HTTPS. Un JSP container fornisce gli stessi servizi di un servlet container ed un motore che interpreta e trasforma una pagina JSP in un servlet. Un Web container fornisce gli stessi servizi di un JSP container e accede alle APIs per i servizi e per la comunicazione di J2EE.

2.1.3 Componenti Enterprise JavaBeans

L'architettura Enterprise JavaBeans è una tecnologia server-side per sviluppare e deployare componenti contenenti la **business logic** di un'applicazione enterprise, cioè la logica che risolve o viene incontro alle necessità di un particolare dominio di business (es. banking, vendita al dettaglio, finanza). I componenti EJB, chiamati **enterprise bean**, sono scalabili, transazionali, e multi-user secure, e girano nel Business Tier. Esistono due tipi di enterprise bean: session bean ed entity bean.

2.1.3.1 Session Bean

Un **session bean** è creato per fornire particolari servizi al client e solitamente esiste solo per la durata di una singola sessione client/server. Un session bean realizza operazioni quali calcoli o accessi ad un database per un client. Sebbene un session bean possa essere transazionale, non è recuperabile se il suo container dovesse crashare. Quindi un session bean rappresenta una conversazione transiente (transient conversation) con un client. Quando il client termina l'esecuzione, il session bean ed i suoi dati sono persi.

I session bean possono essere stateless o stateful; in quest'ultimo caso mantengono uno stato conversazionale (conversational state) attraverso metodi e transazioni. Se mantengono uno stato, l'EJB container gestisce questo stato qualora l'oggetto dovesse essere rimosso dalla memoria. Comunque, l'oggetto stesso session bean deve gestire i propri dati persistenti.

2.1.3.2 Entity Bean

Un **entity bean** è un oggetto persistente che rappresenta i dati memorizzati in una riga di una tabella di un database: il suo focus è data-centric. Può gestire la propria persistenza o delegare questa funzione al suo container. Può vivere tanto quanto i dati che rappresenta.

Un entity bean è univocamente identificato da una **chiave primaria**. Se il container che ospita l'entity bean crasha o il client termina la sua esecuzione, i servizi sottostanti assicurano che l'entity bean, la sua chiave primaria e qualsiasi riferimento remoto sopravvivono e sono salvati.

2.1.3.3 L'EJB Component Container

Gli enterprise bean sono ospitati da un **EJB container**. In aggiunta ai servizi standard, questo fornisce un insieme di servizi per le transazioni e la persistenza e accede alle APIs per i servizi e le comunicazioni in J2EE.

2.1.4 Componenti, Container e Servizi

I tipi di componenti J2EE ed i loro container sono illustrati in Figura 2.1.

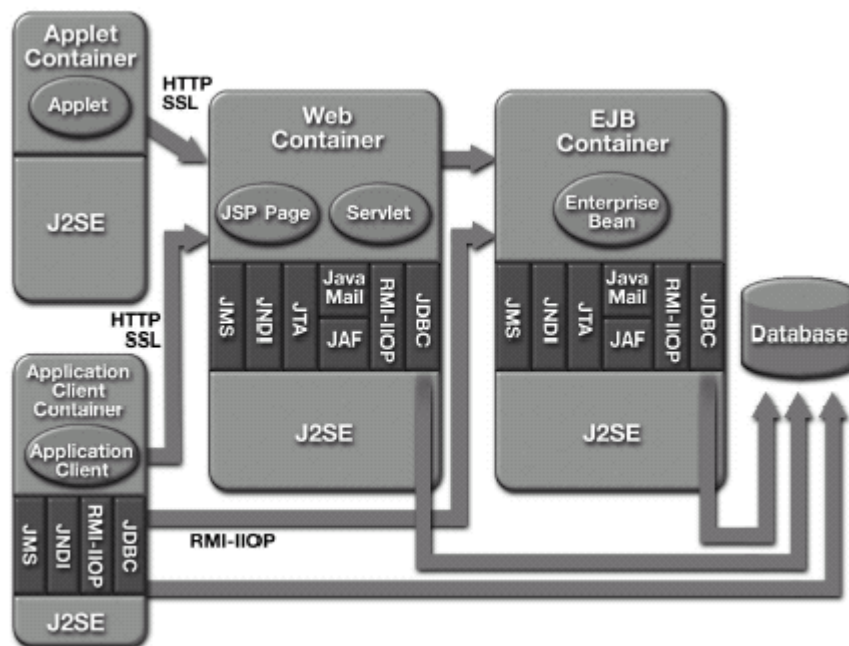


Figura 2.1 Componenti, Container e Servizi nell'architettura J2EE

2.2 I Ruoli nella piattaforma J2EE

La piattaforma J2EE definisce diversi ruoli distinti riguardanti il ciclo di vita di una applicazione. In generale i ruoli sono definiti per aiutare nell'identificazione dei processi realizzati dalle varie parti durante lo sviluppo, il deployment e l'esecuzione di una applicazione J2EE.

- **J2EE Product Provider:** tipicamente un OS vendor, un DBMS vendor, un Application Server vendor, o un Web server vendor; implementa un prodotto J2EE fornendo i container, le APIs ed altre caratteristiche definite nelle specifiche della piattaforma J2EE. Un prodotto J2EE può implementare interfacce non dichiarate o specificate nelle specifiche.

- **Application Component Provider:** produce i blocchi costituenti una applicazione J2EE, cioè componenti Web, enterprise bean, applet, o application client. Ha tipicamente competenze sullo sviluppo di componenti riusabili tanto quanto sufficiente conoscenza di business domain. Un ACP non deve necessariamente conoscere l'ambiente operativo nel quale i componenti saranno usati.
- **Application Assembler:** riceve un file archivio JAR contenente un insieme di componenti sviluppati da ACPs e li assembla a formare il file EAR dell'applicazione J2EE completa. E' competente nel fornire soluzioni per uno specifico problem domain (es. settore finanziario). Gli AAs non devono avere familiarità con il codice sorgente dei componenti con cui hanno a che fare, ma usano dei descrittori per i componenti al fine di conoscere come costruire applicazioni a partire da questi. Come gli ACPs, non devono necessariamente conoscere l'ambiente operativo nel quale le loro applicazioni saranno usate.
- **Deployer:** esperto di uno specifico ambiente operativo, è responsabile del deployment dei componenti e delle applicazioni J2EE all'interno di quell'ambiente. Usa tool forniti dal J2EE Product Provider per realizzare i processi di deployment. Un deployer installa componenti ed applicazioni in un server J2EE e li configura così da risolvere tutte le dipendenze esterne dichiarate dall'ACP e dall'AA.
- **System Administrator:** è il responsabile della configurazione e dell'amministrazione dell'infrastruttura di calcolo e networking dell'azienda nella quale le applicazioni girano, e del controllo e della sorveglianza (overseeing) dell'ambiente runtime. Tipicamente per compiere tali processi usa tool di monitoraggio e gestione runtime forniti dal J2EE Product Provider.
- **Tool Provider:** fornisce tool usati per lo sviluppo ed il packaging dei componenti applicativi.

2.3 Service Technologies

Le Service Technologies di J2EE permettono alle applicazioni di accedere ad un insieme ampio di servizi in maniera uniforme.

2.3.1 Java DataBase Connectivity (JDBC)

L'API Java DataBase Connectivity (JDBC) fornisce **connettività database-independent** tra la piattaforma J2EE ed un ampio insieme di fonti di dati. Nella maggior parte dei casi il data source è un DBMS relazionale, ed i suoi dati sono acceduti usando SQL. In ogni caso, i driver per la tecnologia JDBC possono essere implementati anche per permettere accesso a file system legacy o a Object-Oriented DBMS. Tale tecnologia permette all'Application Component Provider di:

- Realizzare la connessione e l'autenticazione ad un database server
- Gestire le transazioni
- Inviare statement SQL ad un database engine per preprocessing ed esecuzione
- Eseguire stored procedure
- Esaminare e modificare i risultati di statement SELECT
- Ottenere meta-informazioni relativamente al database o le entità che lo compongono

L'API JDBC è basata sulle **specifiche X/Open SQL CLI**, su cui si basa anche lo standard ODBC di Microsoft. La differenza tra le due tecnologie sta nel fatto che mentre ODBC rappresenta una serie di C-Level API, JDBC fornisce uno strato di accesso verso database completo e soprattutto completamente ad oggetti.

La JDBC 3.0 API si divide in due package: **java.sql** e **javax.sql**. Entrambi sono inclusi nelle piattaforme J2SE e J2EE.

2.3.1.1 Stabilire una Connessione

L'API JDBC definisce l'interfaccia **Connection** per rappresentare una fonte dati. In uno scenario tipico, un'applicazione JDBC si conatterà ad una fonte dati usando uno dei due meccanismi indicati di seguito.

- **DriverManager**, una classe completamente implementata che fu introdotta nell'originale JDBC 1.0 API e richiede il caricamento di uno specifico driver
- **DataSource**, un'interfaccia introdotta nel JDBC 2.0 Opzional Package API. E' preferibile al meccanismo DriverManager perché permette di mantenere trasparenti all'applicazione i dettagli sulla fonte dati sottostante. Le proprietà di un oggetto DataSource sono settate per rappresentare un particolare data source. Quando è invocato il suo metodo **getConnection**, l'istanza DataSource ritornerà una connessione alla fonte dati. Una applicazione può essere diretta ad una fonte dati differente semplicemente cambiando le proprietà dell'oggetto DataSource: non è necessaria alcuna modifica al codice. Parimenti, l'implementazione di un DataSource può essere cambiata senza cambiare il codice dell'applicazione che la usa.

L'API JDBC definisce anche due importanti estensioni dell'interfaccia DataSource a supporto delle applicazioni enterprise. Queste estensioni sono le seguenti due interfacce:

- **ConnectionPoolDataSource** supporta il caching ed il riutilizzo di connessioni fisiche, migliorando la performance e la scalabilità dell'applicazione
- **XADataSource** fornisce connessioni che possono partecipare in transazioni distribuite

2.3.1.2 Eseguire statement SQL e manipolare i risultati

Una volta stabilita una connessione, un'applicazione può eseguire query ed aggiornare dati. I metodi dell'interfaccia Connection sono usati per specificare attributi di una transazione e creare oggetti **Statement**, **PreparedStatement** o **CallableStatement**. Questi oggetti vengono usati per eseguire statement SQL e recuperare i risultati. L'interfaccia **ResultSet** incapsula i risultati di una query SQL.

2.3.1.3 Il Modello 2-tier

Il modello 2-tier divide le funzionalità in uno strato client ed uno strato server, come mostrato in Figura 2.2.

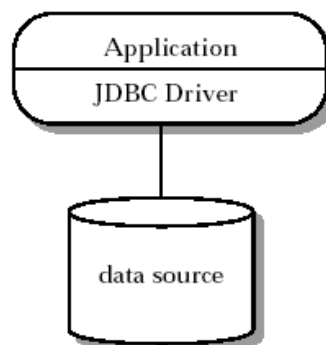


Figura 2.2 Modello 2-tier di JDBC

Lo strato client include l'applicazione e uno o più driver JDBC. L'applicazione gestisce direttamente la presentation logic, la business logic, le transazioni e le risorse, ed interagisce direttamente con i driver JDBC. Tale modello presenta alcuni limiti. Innanzi tutto unisce la presentation e la business logic con le funzioni system-level fornite dall'infrastruttura, cosa che complica la manutenzione del codice; in secondo luogo rende le applicazioni meno portabili perché le lega ad un particolare database; infine limita la scalabilità.

2.3.1.4 Il Modello 3-tier

Il modello 3-tier introduce un middle-tier server che gestisce la business logic e l'infrastruttura, come mostrato nella Figura 2.3 alla pagina seguente.

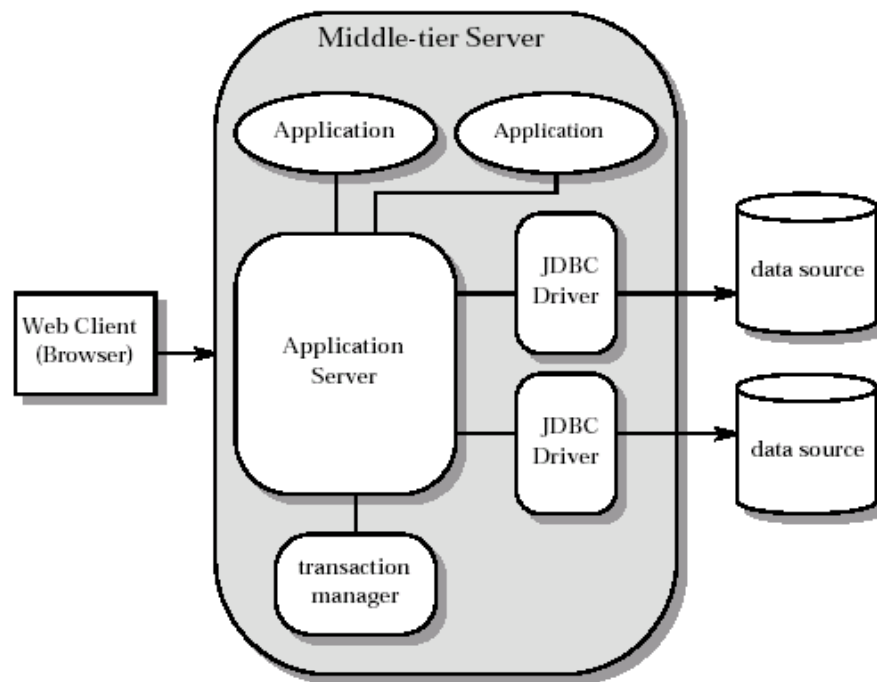


Figura 2.3 Modello 3-tier di JDBC

Questa architettura è disegnata per fornire migliori performance, scalabilità e disponibilità alle applicazioni enterprise. Le funzionalità sono divise tra i diversi tier come segue:

1. Il **Client Tier** è uno thin layer che implementa la presentation logic per l'interazione con l'utente. Programmi Java, browser Web e PDAs sono tipiche implementazioni client-tier. Il client interagisce con l'applicazione middle-tier e non deve includere alcuna conoscenza dell'infrastruttura o dei data source sottostanti.
2. Il **Middle-tier Server** include:
 - **Applicazioni** per interagire con il client ed implementare la business logic.
 - Un **Application Server** (AS) per fornire ad un largo numero di applicazioni supporto alla infrastruttura, tra cui gestione e pooling delle connessioni fisiche e delle transazioni, e mascherare le differenze tra i diversi driver JDBC. Ciò facilita la scrittura di applicazioni portabili.
L'Application Server può essere implementato da un J2EE Server. Gli Application Server implementano le astrazioni di più alto livello usate dalle applicazioni e interagiscono direttamente con i driver JDBC.
 - **Driver JDBC**, i quali forniscono la connettività ai data source sottostanti. Ciascun driver implementa lo standard JDBC API.
3. I **data source** sottostanti dove risiedono i dati. Questo tier può includere RDBMS, file system legacy, OODBMS, data warehouse.

2.3.1.5 JDBC e la piattaforma J2EE

Componenti come JSP, Servlet ed EJB spesso richiedono accesso a dati relazionali, e usano l'API JDBC a tale scopo. In questo caso il container gestisce transazioni e data source. Ciò significa che gli sviluppatori di componenti non usano direttamente i servizi di gestione delle transazioni e dei data source forniti dall'API JDBC.

Segue un semplice esempio di una pagina JSP che estrae i dati da un database e li visualizza in una tabella HTML. L'esempio si riferisce ad un semplice database MsAccess: è per questo usato un driver JDBC-ODBC. Si suppone di aver creato un'origine dati ODBC per tale database dal nome "Agenda".

```

<%@ page language="java"%>
<HTML>
<HEAD><TITLE>Agenda</TITLE></HEAD>
<BODY>
<%
Connection con;
try {
    Class.forName("sun.jdbc.odbc.JdbcOdbcDriver"); // carica la classe e crea un'istanza del driver JDBC-ODBC
}
catch (ClassNotFoundException e) {
    System.out.println("Impossibile caricare il driver: "+ e);
}
try {
    con=DriverManager.getConnection("Jdbc:Odbc:Agenda"); // crea una connessione al DB
}
catch (SQLException e) {
    System.out.println("Impossibile caricare il driver: "+ e);
}
try {
    Statement st=con.createStatement(); // crea uno Statement
    ResultSet rs=st.executeQuery("SELECT * FROM Anagrafica"); // esegue la query ottenendo un ResultSet
}
catch (SQLException e) {
    System.out.println("Errore SQL: "+ e);
}
%>
    <TABLE>
        <TR>
            <TD>Cod</TD>
            <TD>Nome</TD>
            <TD>Cognome</TD>
            <TD>Telefono</TD>
        </TR>
<%
try {
    while(rs.next()) { // si sposta alla riga successiva del ResultSet rs se esiste, altrimenti false
        // aggiunge una riga alla tabella HTML per ogni record presente nella tabella Anagrafica
    %>
        <TR>
            <TD><%=rs.getString(1)%></TD>
            <TD><%=rs.getString(2)%></TD>
            <TD><%=rs.getString(3)%></TD>
            <TD><%=rs.getString(4)%></TD>
        </TR>
<%
    } // fine while
}
catch (SQLException e) {
    System.out.println("Errore SQL: "+ e);
}
st.close(); // rilascia lo Statement
con.close(); // si disconnette dal DB
%>
    </TABLE>
</BODY>
</HTML>

```

Esempio 2.1 Una pagina JSP che utilizza l'API JDBC

2.3.2 Java Transaction API (JTA) e Java Transaction Service (JTS)

Java Transaction API (JTA) permette alle applicazioni di accedere alle transazioni in maniera del tutto indipendente dalla specifica implementazione. JTA specifica interfacce Java standard tra un transaction manager e le parti coinvolte in un sistema transazionale distribuito: l'applicazione transazionale, il J2EE server ed il resource manager che controlla l'accesso alle risorse condivise da parte delle transazioni. **Java Transaction Service (JTS)** specifica l'implementazione di un transaction manager che supporta JTA ed implementa il mapping Java delle Object Management Group Object Transaction Service 1.1 specification. Un transaction manager JTS fornisce i servizi e le funzioni di gestione richiesti per supportare funzionalità specifiche di una particolare istanza di transazione: transaction demarcation, transactional resource management, synchronization e propagation of information.

2.3.3 Java Naming and Directory Interface (JNDI)

L'API **Java Naming and Directory Interface** è una estensione standard della piattaforma Java che fornisce accesso a servizi eterogenei di naming e directory (attraverso la rete). Fornisce alle applicazioni metodi per realizzare operazioni standard, come associare attributi agli oggetti, ottenere oggetti o dati tramite il loro nome, ricercare oggetti o dati mediante l'uso di attributi a loro associati. Ad esempio, tramite JNDI è possibile accedere ad informazioni relative ad utenti di rete, server o workstation, sottoreti o servizi generici, come mostrato in Figura 2.4.

Usando JNDI un'applicazione può memorizzare e reperire qualsiasi tipo di oggetto Java che ha un nome. Come per JDBC, JNDI non nasce per fornire accesso in modo specifico ad un particolare servizio, ma costituisce un set generico di strumenti in grado di interfacciarsi a servizi mediante driver rilasciati dal produttore del servizio, i quali mappano l'API JNDI nel protocollo proprietario di ogni specifico servizio. Tali driver sono chiamati **Service Provider** e forniscono accesso a protocolli come LDAP, NIS, Novell NDS oltre che ad una gran quantità di servizi come DNS, RMI o CORBA Registry. Questo permette alle applicazioni di coesistere con applicazioni e sistemi legacy.

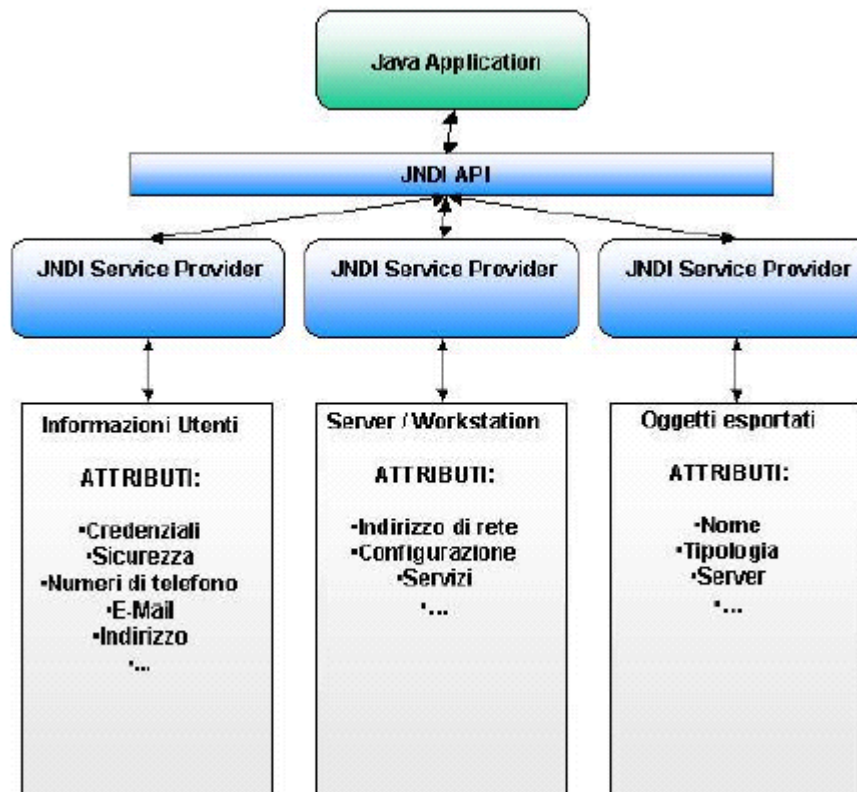


Figura 2.4 Utilizzo di Java Naming and Directory Interface

2.3.4 J2EE Connector Architecture (JCA)

Come già discusso, l'integrazione con gli EISs esistenti è un aspetto molto importante. Le aziende impegnate in un e-business di successo devono integrare gli EISs esistenti con le applicazioni Web-based. Tali aziende hanno bisogno di estendere la portata dei loro EIS per supportare transazioni business-to-business (B2B). Prima della definizione della J2EE Connector Architecture, nessuna specifica per la piattaforma Java era indirizzata a fornire una architettura standard per l'integrazione di EISs eterogenei. La maggior parte degli EIS vendor e degli AS vendor usano architetture non-standard vendor-specific per fornire connettività tra i loro Application Server e gli EISs. La Figura 2.5 mostra la complessità di un ambiente eterogeneo.

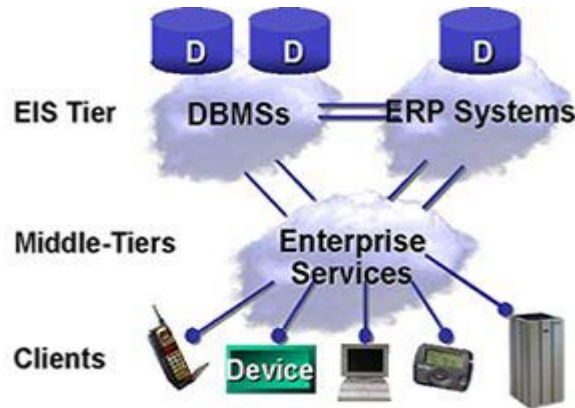


Figura 2.5 Complessità di un ambiente eterogeneo

La **J2EE Connector Architecture** fornisce una soluzione a tale problema. Usando JCA gli EIS vendor non devono più personalizzare i loro prodotti per ogni Application Server, così come gli Application Server vendor non hanno bisogno di aggiungere codice customizzato per connettersi ad un nuovo EIS. Una versione futura della piattaforma J2EE supporterà la J2EE Connector Architecture.

La Connector Architecture definisce un'architettura standard per connettere la piattaforma J2EE ad EISs eterogenei, come sistemi di Enterprise Resource Planning (ERP), mainframe Transaction Processing (TP), database ed applicazioni legacy non scritte in linguaggio Java. L'architettura definisce un insieme di meccanismi scalabili, sicuri e transazionali che descrivono l'integrazione degli EISs con un EJB server (o Application Server) e con applicazioni enterprise.

La J2EE Connector Architecture permette a ciascun EIS vendor di fornire un resource adapter standard per il proprio EIS. Un **resource adapter** è un driver software system-level usato dalle applicazioni J2EE per connettersi ad un EIS. Il resource adapter è inserito in un Application Server e fornisce connettività tra l'EIS, l'Application Server, e l'applicazione enterprise.

Una volta esteso il suo sistema per supportare la J2EE Connector Architecture, un Application Server vendor è sicuro di avere connettività verso molteplici EISs. Allo stesso modo, se un EIS vendor fornisce un resource adapter standard ha la capacità di fornire connettività standard a ciascun Application Server che supporti la JCA.

La Figura 2.6 alla pagina seguente mostra che un EIS resource adapter standard può connettere in maniera standard molteplici Application Server a quel tipo di EIS. Similmente, molteplici resource adapter da diversi EISs possono permettere connettività standard ad un Application Server verso i diversi EIS. Questa system-level pluggability è resa possibile dalla Connector Architecture.

Se ci sono m Application Server ed n EISs, la Connector Architecture riduce la complessità del problema dell'integrazione da un problema $m \times n$ ad un problema $m + n$.

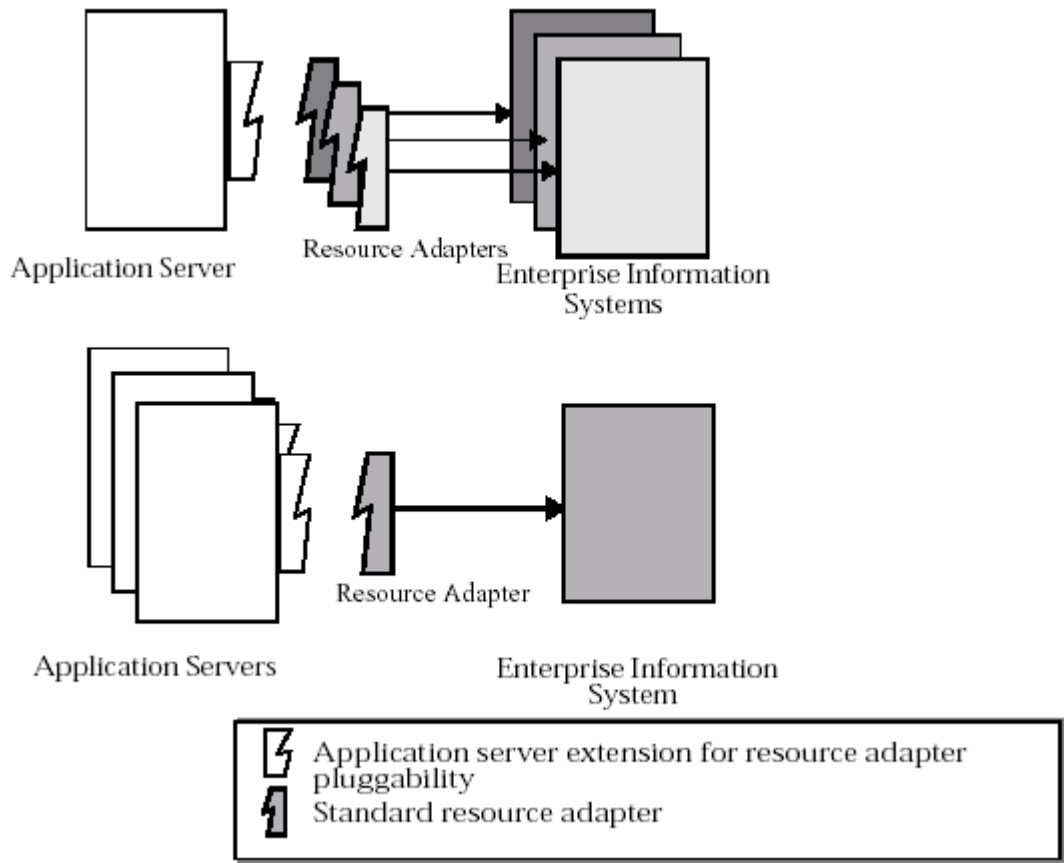


Figura 2.6 Pluggability System-Level tra Application Server ed EISs

2.4 Communication Technologies

Le J2EE Communication Technologies forniscono i meccanismi per la comunicazione tra client e server e tra oggetti collaboranti residenti su server differenti. Le specifiche richiedono supporto per i seguenti tipi di tecnologie per la comunicazione:

- Protocolli Internet
- Protocolli Remote Method Invocation (RMI)
- Protocolli Object Management Group (OMG)
- Tecnologie di Messaging
- Data formats

2.4.1 Protocolli Internet

Definiscono gli standard grazie ai quali parti diverse della piattaforma J2EE comunicano tra loro e con entità remote. La piattaforma J2EE supporta i seguenti protocolli Internet:

- **TCP/IP - Transport Control Protocol over Internet Protocol.** Questi due protocolli sono il componente centrale della comunicazione su Internet e permettono la consegna affidabile di stream di dati da un host ad un altro.

IP, il protocollo di base di Internet, fornisce una consegna non affidabile (unreliable delivery) di singoli pacchetti da un host all'altro. Non dà quindi garanzie sulla consegna del pacchetto, su quanto ci impiegherà, o se i pacchetti arriveranno nello stesso ordine di partenza. Il TCP aggiunge i concetti di connessione ed affidabilità (reliability).

Il protocollo TCP, appartenente allo strato di trasporto del TCP/IP, trasmette dati da server a client suddividendo un messaggio di dimensioni arbitrarie in frammenti (o **datagrammi**) da spedire separatamente e non necessariamente sequenzialmente, per ricomporli nell'ordine corretto alla consegna. Può essere richiesta una nuova trasmissione per un pacchetto non

arrivato a destinazione o contenente dati affetti da errori. Tutto questo in maniera del tutto trasparente rispetto alle applicazioni che trasmettono e ricevono il dato.

A tal fine, TCP stabilisce un collegamento logico (o **connessione**) tra il computer mittente ed il computer destinatario, creando una sorta di canale attraverso il quale le due applicazioni possono inviarsi dati in forma di pacchetti di lunghezza arbitraria. Per questo motivo TCP fornisce un servizio di trasporto affidabile garantendo una corretta trasmissione dei dati tra applicazioni.

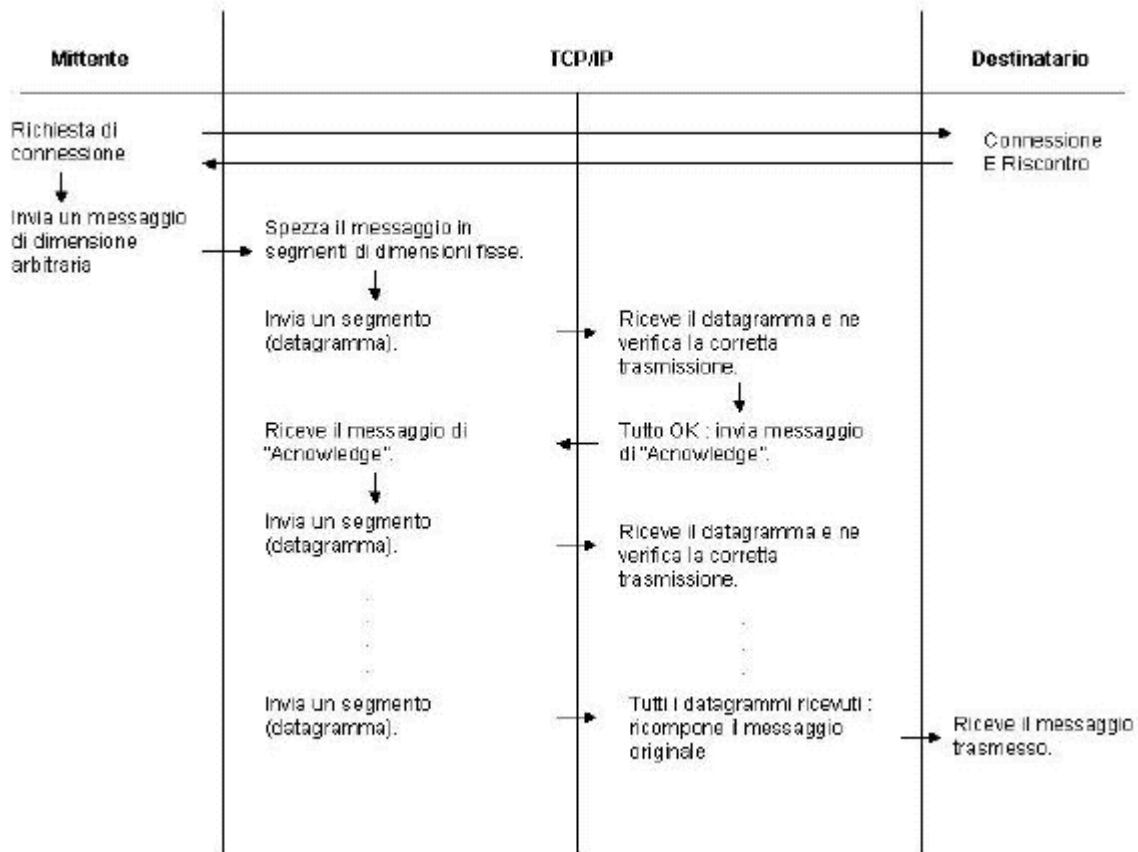


Figura 2.7 Time-line di una trasmissione Connection-oriented

Nella Figura 2.7 è riportato in maniera schematica il flusso di attività che il TCP/IP deve eseguire in caso di trasmissioni di questo tipo. Queste operazioni, soprattutto su reti di grandi dimensioni, risultano essere molto complicate ed estremamente gravose, in quanto comportano che, per assolvere il suo compito, il TCP debba tener traccia di tutti i possibili percorsi tra mittente e destinatario.

Un'altra capacità del TCP garantita dalla trasmissione in presenza di una connessione è quella di poter bilanciare la velocità di trasmissione tra mittente e destinatario. Tale capacità risulta molto utile, particolarmente nel caso in cui la trasmissione avvenga in presenza di reti eterogenee. In generale, quando due sistemi comunicano tra di loro, è necessario stabilire le dimensioni massime che un datagramma può raggiungere affinché possa esservi trasferimento di dati. Tali dimensioni sono dipendenti dalla infrastruttura di rete, dal sistema operativo della macchina host e, possono quindi variare anche per macchine appartenenti alla stessa rete.

Quando viene stabilita una connessione tra due host, il TCP/IP negozia le dimensioni massime per la trasmissione dei dati in ciascuna direzione. Mediante il meccanismo di accettazione di un datagramma (**acknowledge**) viene trasmesso di volta in volta un nuovo valore di dimensione (**finestra**) che il mittente potrà utilizzare nell'invio del datagramma successivo. Questo valore può variare in maniera crescente o decrescente a seconda dello stato della infrastruttura di rete.

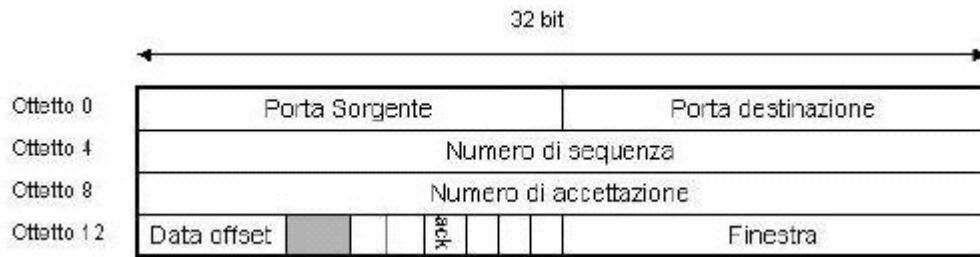


Figura 2.8 I primi 16 ottetti di un segmento TCP

Nonostante le sue caratteristiche, il TCP/IP non risulta essere sempre la scelta migliore relativamente al metodo di trasmissione di dati. Dovendo fornire tanti servizi, il protocollo in questione oltre ai dati relativi al messaggio da trasmettere deve trasportare una quantità informazioni a volte non necessarie. Tali informazioni, spesso, possono diventare causa di sovraccarico sulla rete. In questi casi, a discapito della qualità della trasmissione, è necessario favorirne la velocità adottando trasmissioni di tipo Connectionless (protocollo **User Datagram Protocol** o UDP).

- **HTTP 1.0 – HyperText Transfer Protocol.** Il protocollo HTTP rappresenta il cuore della comunicazione Web. Si appoggia sul TCP/IP ed è il protocollo che usano client e server Web per comunicare attraverso lo scambio di documenti ipertestuali. Un J2EE Web container deve poter fornire i suoi servizi HTTP sulla porta HTTP standard, la **porta 80**. I messaggi HTTP consistono di richieste dal client al server e risposte dal server al client.

Un client HTTP (es. browser Web) deve principalmente trasmettere al server le richieste di reperimento dati, ricevere dal server le informazioni richieste, visualizzare il contenuto delle informazioni richieste e consentire operazioni locali sulle informazioni ricevute (es. salvare una pagina Web o stamparla).

I principali compiti dell'HTTP server sono invece rimanere in ascolto di richieste da parte dei client e soddisfare le richieste inviando al client il documento richiesto; deve rispondere alla richiesta nel modo più efficiente possibile e gestire molte richieste contemporaneamente.

L'interazione fra client e server avviene in diverse fasi:

- Apertura di una connessione TCP
- Invio di una richiesta da parte del client, che specifica la risorsa desiderata
- Invio di una risposta da parte del server che contiene la risorsa richiesta
- Chiusura della connessione TCP

Il protocollo è di tipo **stateless** poiché non è previsto il concetto di sessione.

La richiesta del client contiene diverse informazioni, tra cui il metodo (cioè il comando) che si chiede al server di eseguire, il numero di versione del protocollo HTTP, l'indicazione dell'oggetto al quale applicare il comando, il tipo di client e i tipi di dati che il client accetta.

I metodi definiti in HTTP sono i seguenti: **GET** (richiesta di ricevere un oggetto), **POST** (richiesta di concatenare sul server un oggetto a un altro), **HEAD** (richiesta di ricevere la sola parte <head> di una pagina), **DELETE**, **LINK** e **UNLINK**, **PUT**.

La risposta del server è composta da: una riga di stato, che specifica l'esito della richiesta e indica la versione del protocollo HTTP, un codice numerico di stato, e la specifica testuale dello stato (es. 404 not found); l'oggetto richiesto; alcune meta-informazioni sul tipo di dati dell'oggetto richiesto, tra le quali **server** (tipo di server), **date** (data e ora della risposta), **content-type** (tipo dell'oggetto), **content-length** (numero di byte dell'oggetto), **content-language** (linguaggio delle informazioni), **content-encoding** (tipo di codifica dell'oggetto).

- **SSL 3.0 – Secure Socket Layer.** E' un protocollo di sicurezza che fornisce privacy su Internet e per le comunicazioni Web. E' disponibile indirettamente quando si usano URL **https**. Un J2EE Web container deve poter fornire i suoi servizi HTTPS sulla porta HTTPS standard, la **porta 443**. Il protocollo permette ad applicazioni client/server di avere una comunicazione non "origliata" (uneavesdropped) o non "manomessa" (untampered). I server sono sempre autenticati e lo possono essere, opzionalmente, anche i client.

2.4.2 Protocolli Remote Method Invocation (RMI)

Remote Method Invocation è un set di APIs che permettono agli sviluppatori di realizzare applicazioni distribuite usando il linguaggio Java. RMI usa interfacce Java per definire oggetti remoti, ed una combinazione della tecnologia Java Serialization e di Java Remote Method Protocol (JRMP) per convertire chiamate a metodi locali in chiamate a metodi remoti. La piattaforma J2EE supporta il protocollo JRMP, il meccanismo di trasporto per comunicazioni tra oggetti Java in spazi di indirizzamento (o degli indirizzi) differenti.

Dunque RMI fornisce il supporto per sviluppare applicazioni Java in grado di **invocare metodi di oggetti distribuiti** su JVM differenti sparse per la rete. Grazie a questa tecnologia è possibile realizzare architetture distribuite in cui un client invoca metodi di oggetti residenti su un server che a sua volta può essere client nei confronti di un altro server.

Oltre a garantire tutti i vantaggi tipici di una architettura distribuita, essendo fortemente incentrato su Java RMI consente di trasportare in ambiente distribuito tutte le caratteristiche di portabilità e semplicità legata allo sviluppo di componenti object-oriented, apportando nuovi e significativi vantaggi rispetto alle precedenti tecnologie distribuite (es. CORBA).

Primo, RMI è in grado di passare oggetti e ritornare valori durante una chiamata a metodo oltre che tipi di dati predefiniti. Questo significa che dati strutturati e complessi come le Hashtable possono essere passati come un singolo argomento, senza dover decomporre l'oggetto in tipi di dati primitivi. In poche parole, RMI permette di trasportare oggetti attraverso le infrastrutture di una architettura enterprise senza necessitare di codice aggiuntivo.

Secondo, RMI consente di delegare l'implementazione di una classe dal client al server o viceversa. Questo fornisce una enorme flessibilità al programmatore che scriverà solo una volta il codice per implementare un oggetto, che sarà immediatamente visibile sia al client che al server.

Terzo, RMI estende le architetture distribuite consentendo l'uso di thread da parte di un server RMI per garantire la gestione ottimale della concorrenza tra oggetti distribuiti.

Infine RMI abbraccia completamente la filosofia "Write Once, Run Anywhere" di Java. Ogni sistema RMI è portabile al 100% su ogni Java Virtual Machine.

RMI fornisce quindi una soluzione ottima come supporto ad oggetti distribuiti con la limitazione che tali oggetti devono essere scritti in Java. Tale soluzione non si adatta invece alle architetture in cui gli oggetti distribuiti siano scritti con linguaggi arbitrari. Per far fronte a tali situazioni, la Sun offre anche la soluzione basata su CORBA per la chiamata ad oggetti remoti.

2.4.3 Protocolli Object Management Group (OMG)

I protocolli OMG permettono ad oggetti residenti nella piattaforma J2EE di accedere ad oggetti remoti sviluppati usando l'architettura standard **Common Object Request Broker Architecture** (CORBA) definita dall'OMG e viceversa. CORBA quindi non è un prodotto ma, appunto, uno standard architetturale che definisce un linguaggio dichiarativo **Interface Definition Language** (IDL), indipendente dalla piattaforma e dal linguaggio di riferimento con cui l'oggetto è stato implementato, per la descrizione delle interfacce degli oggetti, ed un sistema di comunicazione, l'**Object Request Broker** (ORB), che realizza la trasparenza, la localizzazione e l'attivazione degli oggetti. Gli oggetti CORBA sono definiti usando l'IDL. Un ACP definisce l'interfaccia di un oggetto remoto in IDL e successivamente usa un IDL compiler per generare gli stub client e server che connettono le implementazioni dell'oggetto ad un ORB, una libreria, come detto, che permette agli oggetti CORBA di localizzare tutti gli altri oggetti e comunicare con loro. Le ORBs comunicano tra loro usando Internet Inter-ORB Protocol (**IIOP**). Le tecnologie OMG richieste dalla piattaforma J2EE sono **Java IDL** e **RMI-IIOP**.

2.4.4 Tecnologie di Messaging

Le tecnologie di messaging forniscono un modo per inviare e ricevere messaggi in modalità asincrona. La tecnologia **Java Message Service** (JMS) fornisce supporto alle applicazioni enterprise nella gestione asincrona della comunicazione verso servizi di messaging o nella creazione di nuovi MOM (Message Oriented Middleware). Nonostante JMS non sia così largamente diffuso come le altre tecnologie, svolge un ruolo importantissimo nell'ambito di sistemi enterprise.

Per meglio comprenderne il motivo è necessario chiarire il significato della parola **messaggio**, che in ambito JMS rappresenta tutto l'insieme di messaggi asincroni utilizzati dalle applicazioni

enterprise, non dagli utenti umani, contenenti dati relativi a richieste, report o eventi asincroni che si verificano all'interno del sistema, fornendo informazioni vitali per il coordinamento delle attività tra i processi. I messaggi contengono quindi informazioni, impacchettate secondo specifici formati, relative a particolari eventi di business. I messaggi JMS sono usati per coordinare queste applicazioni.

Grazie a JMS è possibile scrivere applicazioni di business message-based altamente portabili, fornendo una alternativa a RMI, necessaria in determinati ambiti applicativi. Nella Figura 2.9 è schematizzata una soluzione tipo di applicazione basata su messaggi. Una applicazione di Home Banking deve interfacciarsi con il sistema di messaggistica bancaria ed interbancaria da cui trarre tutte le informazioni relative alla gestione economica e finanziaria dell'utente.

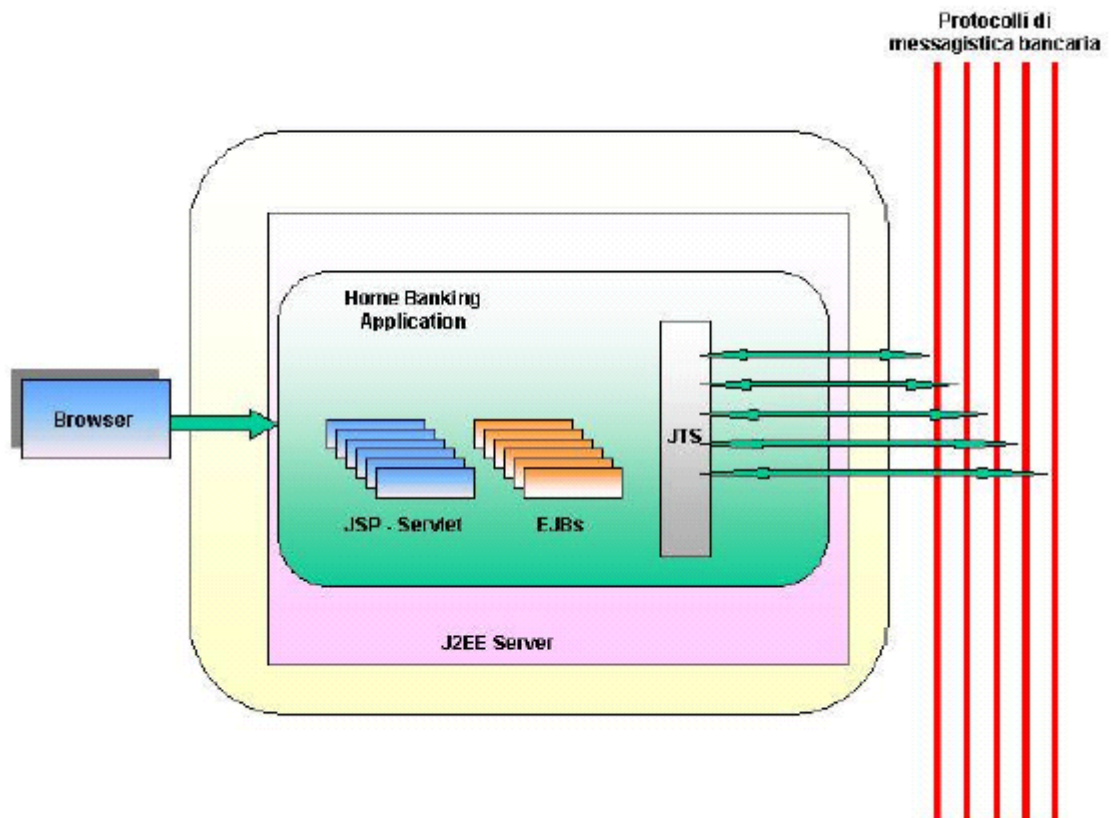


Figura 2.9 Esempio di applicazione message-based

La seconda tecnologia di messaging supportata dalla piattaforma J2EE è JavaMail. L'insieme di API **JavaMail** fornisce un'interfaccia per inviare e ricevere messaggi utilizzati dagli utenti.

Sebbene entrambe le API possano essere usate per notifiche asincrone, JMS è preferita quando velocità ed affidabilità sono requisiti primari.

2.4.5 Data Formats

I data formats definiscono il tipo di dati che possono essere scambiati tra componenti. J2EE richiede supporto per i seguenti formati:

- **HTML 3.2 (HyperText Markup Language):** è il **linguaggio di markup** (cioè i documenti HTML sono file di testo ASCII intercalati da appositi marcatori detti tag, che definiscono il formato del testo) principale usato per definire documenti ipertestuali presenti su Internet. HTML permette di incorporare immagini, suoni, video stream, campi form, collegamenti ipertestuali ad altri documenti HTML e formattazione testuale di base. I documenti HTML hanno una posizione (location) unica su Internet.

I file HTML sono semplici file di testo ASCII che non contengono direttamente informazioni dipendenti dalla piattaforma o da uno specifico programma, e possono essere letti e scritti con un normale editor di testi.

I **tag HTML** sono identificativi di un tipo di formattazione che deve essere applicata al testo contenuti tra < e >. Ogni tipo di tag ha un insieme di possibili attributi che ne specificano l'azione. Quando la formattazione deve essere applicata ad un segmento di testo (ovvero da un certo punto ad un certo altro punto) vengono usati due tag: un tag di inizio e un tag di fine formattazione. Altri tag hanno una azione puntuale, che si esercita in un certo punto del testo, come inserire una immagine (es.). In questo caso non esiste il tag di chiusura. La struttura di un file HTML è la seguente:

```
<HTML>
<HEAD>
...intestazione...
</HEAD>
<BODY>
...contenuto...
</BODY>
</HTML>
```

Il tag **<HEAD>** specifica l'intestazione del file. Contiene solo alcuni tag che commentano o classificano il reale contenuto della pagina. In particolare sono interessanti il tag **<TITLE>** che definisce il titolo visualizzato nella barra della finestra, e i tag **<META>** che inseriscono commenti. Tra i tag **<BODY>** e **</BODY>** è definito tutto il contenuto della pagina.

Utilizzando gli attributi del tag **<BODY>** è possibile definire alcune caratteristiche visibili generali della pagina, quali ad esempio lo sfondo e i colori del testo. HTML usa una codifica dei colori basata sulla scomposizione Red-Green-Blue. Ogni colore è definito da due cifre esadecimali (256 codici per colore base). Ad esempio FFFFFFFF definisce il bianco, 000000 il nero, FF0000 il rosso, 00FF00 il verde, 0000FF il blu.

I **titoli** sono utilizzati per dividere in sezioni il testo. L'HTML definisce sei livelli di titolo con i tag: <H1></H1>.....<H6></H6>. Il testo normale è diviso in **paragrafi** dai tag <P></P>.

Il tag di base per cambiare le caratteristiche del testo è **** (attributi SIZE, COLOR e FACE). I **link ipertestuali** in HTML sono orientati da una sorgente ad una destinazione. E' possibile fare un link oltre che ad altre pagine Web anche ad altre risorse Internet, come per esempio un utente di e-mail, una news o un file ftp. In sostanza quindi la destinazione del link può essere una qualunque URL. Per creare un link HTML usa il tag **<A>** (anchor), che ha un attributo obbligatorio, HREF (abbreviazione di "Hypertext REFERENCE"), usato per specificare l'URL di destinazione. Una **tabella HTML** è una sequenza di righe, ciascuna composta da celle (non sono definite le colonne). Per creare la tabella si usano (innestati) tre diversi tipi di tag: <TABLE></TABLE> definisce la tabella, <TR></TR> definisce le righe nella tabella, <TD></TD> definisce le celle nelle righe. Per tenere separato il contenuto vero e proprio dal layout grafico, dalla versione 4.0 di HTML sono stati introdotti i **fogli di stile** (definiti usando il linguaggio di specifica **Cascading Style Sheet** o CSS). La definizione degli stili va inserita tra i tag <STYLE> e </STYLE> nell'header della pagina.

Il tag HTML per il data input è <FORM></FORM> i cui attributi tipici sono **action**="URL" e **method**="GET"|"POST". Il form consente di raccogliere dati di diversi tipi e con diversi stili di interazione (text, checkbox, radiobutton, textarea, menu). I dati raccolti con il form devono essere mantenuti in associazioni nome=valore. Ogni elemento del form deve quindi avere un nome. Il programma che riceverà i dati da elaborare, ricostruirà le strutture dati. Per spedire il contenuto del form ad un programma su server si deve clickare su un bottone di submit, inserito utilizzando il tag seguente: <input type="submit" value="Click me">.

■ **File di immagine:** la piattaforma J2EE supporta due formati per immagini, GIF e JPEG.

GIF (Graphics Interchange Format) è un protocollo per la trasmissione e lo scambio online di dati grafici raster. GIF è un formato per immagini a 256 colori. Utilizza una tecnica di compressione senza perdita di informazioni (lossless) e si presta maggiormente alla riproduzione di icone, e altre immagini disegnate. Permette di creare immagini animate e di avere effetti di trasparenza.

JPEG (Joint Photographic Experts Group) è uno standard per la compressione di immagini gray-scale o color still. JPEG è un formato grafico che utilizza una tecnica di compressione molto efficace ma con perdita di informazioni. È particolarmente adatto alla presentazione e all'invio di immagini fotografiche digitalizzate poiché opera a 16 Milioni di colori.

- **File JAR (Java ARchive):** JAR è un formato di file platform-independent che permette l'aggregazione di molti file in uno unico.
- **File .class:** il formato di un file Java compilato come specificato nella Java Virtual Machine Specification. Ciascun file class contiene o una classe o una interfaccia Java.
- **XML (eXtensible Markup Language):** è un metalinguaggio di markup (ossia fornisce una grammatica con cui descrivere linguaggi di markup) text-based, sviluppato da XML Working Group con l'obiettivo di disegnare una versione semplificata di SGML (Standard Generalized Markup Language) che consentisse di definire agilmente linguaggi di markup per il Web. Permette di definire il markup necessario per identificare i dati ed il testo nei documenti XML. Analogamente all'HTML, si possono identificare i dati usando tag. Ma diversamente dall'HTML, i tag XML descrivono i dati, piuttosto che il formato per visualizzarli. L'XML fornisce una sintassi con cui è possibile specificare gli elementi e gli attributi che possono essere utilizzati all'interno di particolari classi di documenti. Utilizzando XML è possibile creare un modello, chiamato Document Type Definition (**DTD**), che descrive la struttura e il contenuto di una classe di documenti.

2.5 Riassunto

La piattaforma J2EE si basa su un insieme di tecnologie a componenti (Enterprise JavaBeans, JavaServer Pages e Servlet) che semplificano il processo di sviluppo di applicazioni enterprise. J2EE fornisce un certo numero di servizi system-level che semplifica la programmazione di una applicazione e permette ai componenti di essere customizzati per usare le risorse disponibili nell'ambiente nel quale sono deployate. In aggiunta alle component technologies, la piattaforma J2EE fornisce APIs che permettono ai componenti di accedere ad una varietà di servizi remoti, e meccanismi per la comunicazione tra client e server e tra oggetti collaboranti residenti su server diversi.

Capitolo 3 Un approfondimento sull'architettura MVC

Nell'approccio classico alla programmazione, un programma è tipicamente strutturato in tre momenti:

- Acquisizione dei dati di ingresso - **read**
- Trasformazione dei dati di ingresso al fine di ottenere il risultato - **eval**
- Emissione dei risultati in uscita - **print**

Tale approccio può essere schematizzato scrivendo: **print (eval (read ()))**



L'elaborazione è concentrata nel momento centrale, durante il quale non vi è interazione col mondo esterno. Essa avviene dunque in un mondo chiuso. Ciò rende la computazione più controllabile e prevedibile, ma per questo, intrinsecamente, meno flessibile e meno espressiva.

Questo schema viene radicalmente modificato dal concetto di applicazione dotata di **Graphical User Interface** (o GUI), attraverso la quale l'utente può interagire durante l'elaborazione e determinarne il flusso in modo non prevedibile a priori. Cosa significa "elaborare" in questo scenario? In pratica si svolgono azioni non più in conseguenza del proprio interno flusso di controllo, ma in risposta ad eventi generati dall'esterno.

Il concetto di **evento** introduce quindi un notevole cambiamento nella organizzazione di un sistema software, in quanto implica l'idea di un sistema che "reagisce" a stimoli esterni anziché di un sistema che "decide lui" quando inviare comandi ai dispositivi. Nasce la **programmazione event-driven**: non più algoritmi stile input/elaborazione/output ma reazione agli eventi che l'utente, in modo interattivo, genera sui componenti grafici (concetti di evento e di ascoltatore degli eventi o event listener).

Come strutturare un'applicazione interattiva e guidata da eventi? La risposta a tale domanda fu l'introduzione del **modello di progettazione MVC**, che ebbe origine negli applicativi sviluppati in Smalltalk e secondo il quale conviene modellare un'applicazione interattiva separando:

- la rappresentazione dei dati (**Model**)
- la presentazione dei dati (**View**)
- il comportamento dell'applicazione (**Controller**)

Il **Modello** rappresenta la struttura dei dati nell'applicazione e le relative operazioni (dipendenti dall'applicazione): modella ed elabora il problema che vogliamo risolvere.

La **Vista** rappresenta una "fotografia" dello stato interno del Modello, facilitandone la lettura e l'interpretazione da parte dell'utente umano: presenta i dati all'utente in qualche forma; possono esserci più viste qualora sia utile presentare i dati in modi diversi.

Il **Controllore** controlla il flusso di dati nell'applicazione: è sensibile alle azioni dell'utente e può recuperare i dati forniti, traslando tutto ciò in chiamate di opportuni metodi del Modello e selezionando la Vista appropriata (in base allo stato e alle preferenze dell'utente).

In questo modello la parte eval è sostituita da un oggetto che incapsula l'elaborazione, la parte read è sostituita da uno più oggetti grafici (situati nella finestra associata all'applicazione) che permettono l'immissione di dati prelevabili da parte dell'elaborazione quando questa lo ritiene necessario, e la parte print è sostituita da uno più oggetti grafici (anch'essi situati nella finestra associata) che permettono la visualizzazione di dati da parte della elaborazione.

In più, vi sono uno più oggetti che costituiscono il Modello, vale a dire le entità usate dalla elaborazione, ed uno più oggetti grafici (situati nella finestra associata) che permettono l'invio di eventi i quali, opportunamente gestiti, permettono di attivare la parte di elaborazione.

La Figura 3.1 descrive le relazioni tra Model, View e Controller in una applicazione MVC.

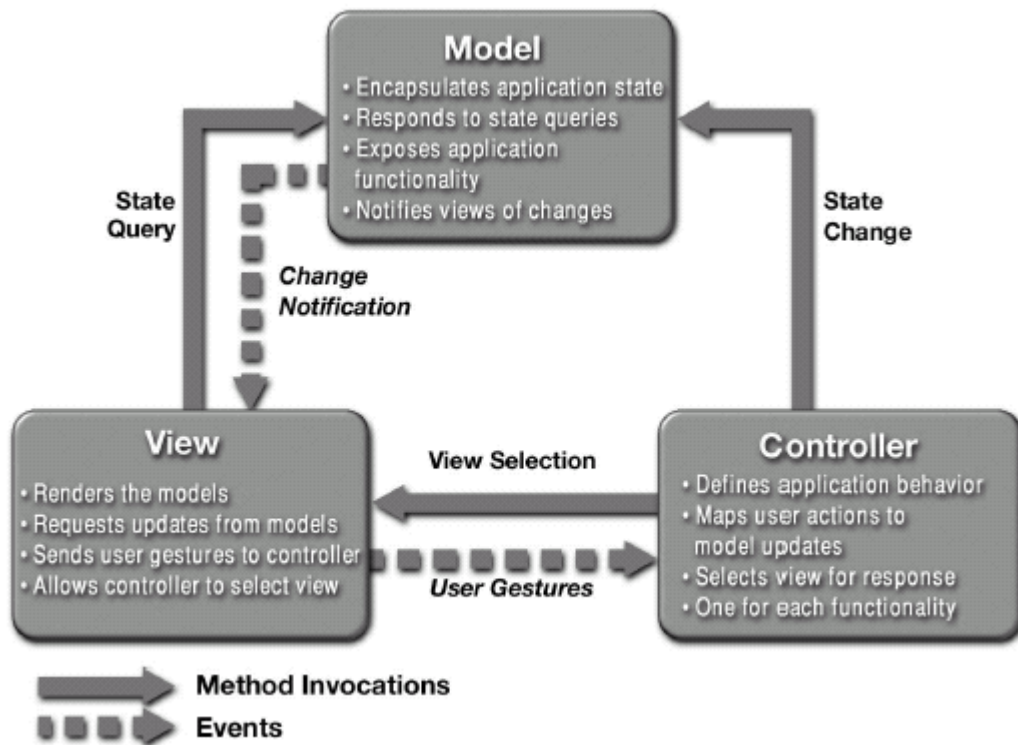


Figura 3.1 Relazione tra Model, View e Controller nell'architettura MVC

Model:

- incapsula lo stato dell'applicazione
- notifica la View dei suoi cambiamenti
- offre la possibilità alla View di ricevere informazioni sul suo stato

View:

- accede ai dati del Model e specifica come questi debbano essere rappresentati
- delega le richieste effettuate dall'utente al Controller
- permette al Controller di selezionare una View

Controller:

- definisce il comportamento dell'applicazione
- interpreta le richieste dell'utente in azioni
- si occupa dei cambiamenti dello stato del Model
- sceglie in base all'azione eseguita la View da mostrare

3.1 Java e l'architettura MVC

L'architettura Java supporta intrinsecamente il concetto di applicazione grafica, mettendo a disposizione due package grafici:

- il package **java.awt** è stato il primo package grafico fornito dalla tecnologia Java (Java 1.0) ed è indipendente dalla piattaforma (...o quasi)
- il package **javax.swing** è il package grafico più recente (Java 2; versione preliminare da Java 1.1.6) ed è realmente indipendente dalla piattaforma

Swing definisce una gerarchia di classi che forniscono ogni tipo di componente grafico: Window (distinte in Frame e Dialog), Panel, Label, List, MenuBar, ScrollBar, TextArea, TextField, Button, CheckBox, RadioButton. Il design pattern MVC è stato utilizzato in Java per la progettazione e lo sviluppo delle componenti AWT/Swing:

- L'utente opera sulla Vista di un programma agendo su una delle sue componenti di controllo (es. bottone).

- Il Controllore è avvertito di tale evento ed esamina la Vista per rilevarne le informazioni aggiuntive.
- Il Controllore invia tali informazioni al Modello che effettua la computazione richiesta e aggiorna il proprio stato interno.
- Il Controllore (o il Modello) richiede alla Vista di visualizzare il risultato della computazione.
- La Vista interroga il Modello sul suo nuovo stato interno, e visualizza l'informazione all'utente.

Diversi sono i vantaggi dell'architettura MVC. Primo, le classi che formano l'applicativo possono essere più facilmente riutilizzate. Secondo, l'applicativo è organizzato in parti più semplici e comprensibili (ogni parte ha le sue specifiche finalità). Infine, la modifica di una parte non coinvolge e non interferisce con le altre parti (maggiore flessibilità nella manutenzione del software).

3.2 L'architettura MVC e le applicazioni distribuite

L'architettura Model-View-Controller è usata per analizzare le caratteristiche delle applicazioni distribuite. Tale astrazione aiuta nel processo di suddivisione dell'applicazione in componenti logiche che possono essere progettate molto più semplicemente (e separatamente). Infatti, l'MVC non è altro che un modello di progettazione (design pattern) che scinde in tre parti l'applicazione distinguendo tra Modello, Vista e Controllore. MVC permette di dividere le funzionalità tra gli oggetti coinvolti nella elaborazione e nella presentazione dei dati così da minimizzare il grado di accoppiamento tra tali oggetti.

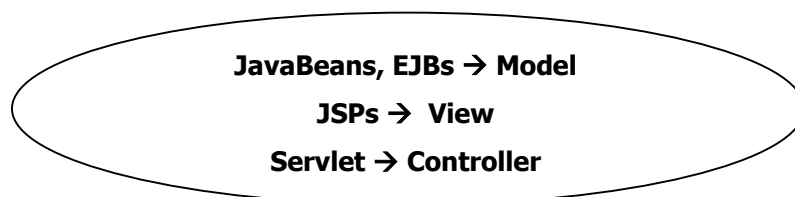
Tale architettura fu sviluppata originariamente, come visto, per adattare i tradizionali processi di input, elaborazione, e output al modello di interazione grafica con l'utente utilizzato nella programmazione event-driven. Il modello MVC viene oggi correttamente utilizzato in fase di progettazione di applicazioni enterprise multi-tier Web-based per integrare gli eventi e la logica con la rappresentazione grafica.

Il Model rappresenta i dati dell'applicazione e le regole business che governano l'accesso e la modifica a tali dati. Il Model notifica alla View quando cambia e le fornisce l'abilità di interrogarlo sul suo stato. Inoltre fornisce l'abilità al Controller di accedere alle funzionalità dell'applicazione incapsulate dal Model.

Una View presenta i contenuti di un Model. Essa accede ai dati del Model e specifica come tali dati dovranno essere presentati. Quando il Model cambia, è responsabilità della View mantenere la consistenza nella sua presentazione. La View inoltra le azioni dell'utente al Controller.

Un Controller definisce il comportamento (behavior) dell'applicazione distribuita: interpreta gli user gesture e li mappa in azioni che devono essere realizzate dal Model.

In una applicazione GUI stand-alone le azioni dell'utente corrispondono a click su pulsanti o menu, mentre in un'applicazione Web le azioni appaiono come richieste HTTP (in GET o POST) del browser al Web Tier. Pertanto, per l'implementazione del Model si ricorre ad una collezione di Java bean e EJB, mentre per l'implementazione del View e del Controller, in una applicazione Web si ricorre a pagine JSP e a Servlet.



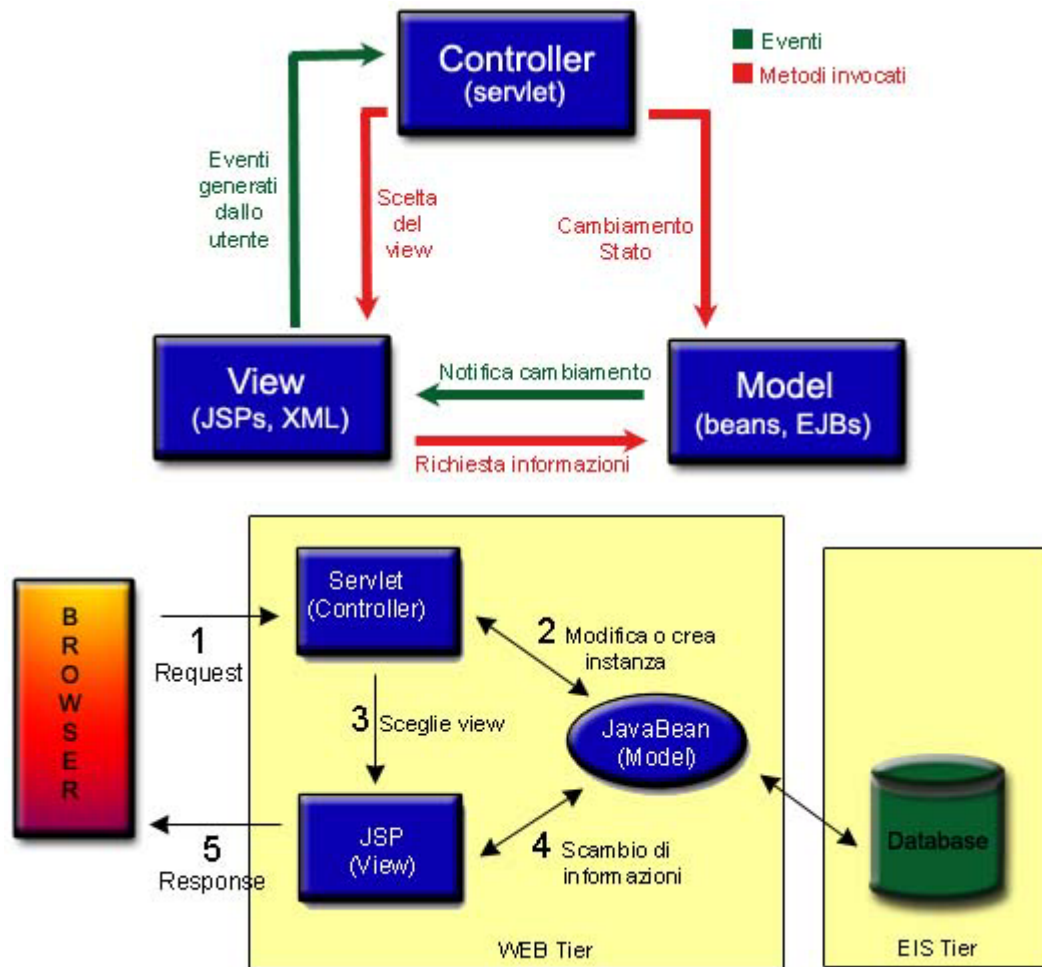


Figura 3.2 MVC in un'applicazione J2EE Web-based

3.3 Un primo esempio

Supponiamo di voler realizzare sul Web un Modello che rappresenti una finestra stile Swing: realizziamo un bean per implementare il Modello, una pagina JSP per eseguire il render della finestra (View) e un servlet per gestire gli eventi di ridimensionamento (Controller).



Figura 3.3 Esempio di utilizzo di MVC

```

public class Window implements java.io.Serializable {
    final private int DEFAULT_X=400;
    final private int DEFAULT_Y=300;
    final private int WINDOW_NORMAL=0;
    final private int WINDOW_ICONIFIED=1;
    final private int WINDOW_CLOSED=2;
    private int x,y;
    private int state;
    public Window( ) { // costruttore della classe
        x = DEFAULT_X;
        y = DEFAULT_Y;
        state = WINDOW_NORMAL;
    }
    public int getX( ) {
        return x;
    }
    public int getY( ) {
        return y;
    }
    public boolean isIconified( ) {
        return (WINDOW_ICONIFIED==state);
    }
    public void setIconified( ) {
        state=WINDOW_ICONIFIED;
        setSize(0,0);
    }
    public boolean isNormalized( ) {
        return (WINDOW_NORMAL==state);
    }
    public void setNormalized( ) {
        state=WINDOW_NORMAL;
        setSize(DEFAULT_X, DEFAULT_Y);
    }
    public boolean isClosed( ) {
        return (WINDOW_CLOSED==state);
    }
    public void setClosed( ) {
        state=WINDOW_CLOSED;
    }
    public void setSize(int x, int y) {
        this.x = x;
        this.y = y;
    }
} // classe Window

```

Esempio 1.1 Il file Window.java

Il file Window.java è un bean costituito da attributi per mantenere lo stato della finestra e da una serie di metodi usati dal View per ricevere informazioni sullo stato della finestra.

```

import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
public class WindowController extends HttpServlet {
    String pageURL="/windows.jsp";
    public void service(HttpServletRequest req, HttpServletResponse res) throws ServletException, IOException {
        Window window=null;
        /* Cerca il bean windowName in sessione altrimenti lo crea */
        String windowName = req.getParameter("windowName");

```

```

    if (windowName!=null) window = (Window) req.getSession( ).getAttribute(windowName);
    if (window==null && windowName!=null) window=doInstance(req, windowName);
    /* Interpreta ed esegue l'azione */
    String action = req.getParameter("action");
    if (action!=null && window!=null) runAction(window, action, req, res);
    /* Delega la presentazione al view */
    getServletContext( ).getRequestDispatcher(pageURL).forward(req,res);
}
private void runAction(Window window, String action, HttpServletRequest req, HttpServletResponse res) {
    if (action.equalsIgnoreCase("iconify")) iconify(window);
    if (action.equalsIgnoreCase("normalize")) normalize(window);
    if (action.equalsIgnoreCase("close")) close(window);
    if (action.equalsIgnoreCase("resize")) resize(window, req, res);
}
private Window doInstance(HttpServletRequest req, String windowName) {
    Window window=new Window( );
    req.getSession( ).setAttribute(windowName, window);
    return window;
}
public void iconify(Window window) {
    window.setIconified( );
}
public void normalize(Window window) {
    window.setNormalized( );
}
public void close(Window window) {
    window.setClosed( );
}
public void resize(Window window, HttpServletRequest req, ServletResponse res) {
    int x,y;
    try {
        x=Integer.parseInt(req.getParameter("x"));
        y=Integer.parseInt(req.getParameter("y"));
        window.setSize(x,y);
    }
    catch (NumberFormatException nfe) { /*Errore: le dimensioni non sono numeriche*/ }
}
} // classe WindowController

```

Esempio 1.2 Il file WindowController.java

Il file WindowController è il servlet incaricato di ricevere ed interpretare le richieste dal browser. Potendo esserci più di una finestra, per prima cosa il servlet individua la finestra soggetta ad un evento leggendo il parametro windowName, quindi preleva dalla sessione il bean di nome windowName, che è una istanza della classe Window, ed esegue su di esso le azioni richieste. Infine dovrebbe decidere quale View mostrare, effettuando il forward alla pagina JSP esatta, ma questo semplice esempio è costituito da una sola View e pertanto esegue sempre un forward alla pagina windows.jsp.

```

<%@ page language="java" contentType="text/html" %>
<jsp:useBean id="window1" scope="session" class="Window" />
<html>
<head><title>MVC Pattern</title></head>
<body>
<% if (!window1.isClosed()) { %>
<table border="2" width="<jsp:getProperty name="window1" property="x" />" height="<jsp:getProperty
name="window1" property="y" />">
<tr>
<td align="right" bgColor="#CCCCCC" height="10">

```

```

<table>
<tr>
<td>
<form action="/mvc/servlet/windows" method="post">
<input type="hidden" name="windowName" value="window1">
<input type="hidden" name="action" value="iconify">
<input type="submit" value="Iconifica">
</form>
</td>
<td>
<form action="/mvc/servlet/windows" method="post">
<input type="hidden" name="windowName" value="window1">
<input type="hidden" name="action" value="normalize">
<input type="submit" value="Ripristina">
</form>
</td>
<td>
<form action="/mvc/servlet/windows" method="post">
<input type="hidden" name="windowName" value="window1">
<input type="hidden" name="action" value="close">
<input type="submit" value="Chiudi">
</form>
</td>
</tr>
</table>
</td>
</tr>
<% if (!window1.isIconified()) { %>
<tr><td><br><br>Questo è il contenuto della finestra.<br><br></td></tr>
<tr>
<td align="right" bgcolor="#CCCCCC" height="10">
<form action="/mvc/servlet/windows" method="post">
<table>
<tr>
<td>X
<input type="text" name="x" value="<jsp:getProperty name="window1" property="x" />" size="3">
</td>
<td>Y
<input type="text" name="y" value="<jsp:getProperty name="window1" property="y" />" size="3">
</td>
<td>
<input type="hidden" name="windowName" value="window1">
<input type="hidden" name="action" value="resize">
<input type="submit" value="Ridimensiona">
</td>
</tr>
</table>
</form>
</td>
</tr>
<% } %>
</table>
<% } %>
</body>
</html>

```

Esempio 1.3 Il file windows.jsp

Come si può vedere la pagina `windows.jsp` risulta contenere poco codice script al suo interno. Si è raggiunta un'ottima separazione della business logic dalla presentation logic. I vantaggi così ottenuti sono i seguenti:

- Suddivisione dei compiti e conseguente distinzione del team di sviluppo in programmatori professionisti e Web developer.
- Riduzione del tempo di sviluppo, soprattutto in caso di ampliamento di funzionalità.
- Maggiore facilità nella manutenzione e nel rinnovamento della veste grafica.

Naturalmente il pattern MVC non funziona come una bacchetta magica; i risultati migliori si ottengono con una buona progettazione e soprattutto prestando attenzione caso per caso. In realtà è possibile anche perfezionare e raffinare i vantaggi così ottenuti: si possono estendere i tag di JSP scrivendo una tag library ad hoc, e in questo esempio, addirittura, è possibile annullare completamente il codice script all'interno della pagina JSP rendendola anche più compatta e leggibile. Ciò sarà visto nel dettaglio nella prossima sezione.

3.4 Miglioriamo l'esempio: utilizzo delle tag library di JSP

In questa sezione si evolve l'esempio trattato in precedenza, continuando ad usare il modello MVC, ma aggiungendo una potente caratteristica della tecnologia JSP, e cioè le librerie di tag estesi (tag library). La sezione precedente ha mostrato come implementare il design pattern MVC usando un bean, un servlet e una pagina JSP. Si supponga di voler semplificare la pagina JSP. L'idea più idonea che salta subito in mente è di estrapolare più codice possibile dalla pagina ed includerlo da qualche parte in una classe Java. La perfezione si raggiungerebbe se si riuscisse a produrre il solito esempio della finestra scrivendo una semplice pagina JSP come quella mostrata di seguito.

```
<%@ page language="java" %>
<html>
<body>
<window-tag bgColor="white" borderColor="gray" width="250" height="150">
contenuto della finestra
</window-tag>
</body>
</html>
```

Esempio 1.4 Esempio di pagina JSP senza codice script

Ebbene, ciò è perfettamente possibile! Per prima cosa occorrerà definire una classe che implementi l'interfaccia **Tag** o più semplicemente che estenda la classe **TagSupport** del package **javax.servlet.jsp.tagext**. I metodi che occorrerà tenere in considerazione sono **doStartTag()** e **doEndTag()** che dovranno essere ridefiniti. Ecco l'esempio pratico:

```
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.*;
public class WindowTag extends TagSupport {
    private Window window;
    private String name;
    private String bgColor;
    private String color;
    private String borderColor;
    private int width;
    private int height;
    private int border;
    private String align;
    public void setName(String str) {name=str;}
    public String getName() {return name;}
    public void setBgColor(String str) {bgColor=str;}
    public String getBgColor() {return bgColor;}
```

```

public void setColor(String str) {color=str;}
public String getColor( ) {return color;}
public void setBorderColor(String str) {borderColor=str;}
public String getBorderColor( ) {return borderColor;}
public void setWidth(int w) {width=w;}
public int getWidth( ) {return width;}
public void setHeight(int h) {height=h;}
public int getHeight( ) {return height;}
public void setBorder(int b) {border=b;}
public int getBorder( ) {return border;}
public void setAlign(String str) {align=str;}
public String getAlign( ) {return align;}
public int doStartTag( ) {
    try {
        window = (Window) pageContext.getSession().getAttribute(name);
        if (window==null) {
            // si crea l'oggetto se non esiste
            window=new Window( );
            window.setDefaultX(width);
            window.setDefaultY(height);
            window.setSize(width, height);
            pageContext.getSession( ).setAttribute(name, window);
        } // fine if
        if (window.isClosed( )) return SKIP_BODY;
        JspWriter out = pageContext.getOut( );
        out.println("<table cellspacing=\"0\" cellpadding=\"0\" borderColor=\"\"+borderColor+\"\"
border=\"\"+border+\"\" bgcolor=\"\"+bgColor+\"\" width=\"\"+window.getX()+\"\">");
        out.println("<tr><td bgcolor=\"\"+borderColor+\"\">");
        out.println("<table align=\"right\"><tr><td>");
        out.println("<form action=\"windows\" method=\"post\">");
        out.println("<input type=\"hidden\" name=\"windowName\" value=\"\"+name+\"\">");
        out.println("<input type=\"hidden\" name=\"action\" value=\"iconify\">");
        out.println("<input type=\"submit\" value=\"Iconifica\">");
        out.println("</form></td><td>");
        out.println("<form action=\"windows\" method=\"post\">");
        out.println("<input type=\"hidden\" name=\"windowName\" value=\"\"+name+\"\">");
        out.println("<input type=\"hidden\" name=\"action\" value=\"normalize\">");
        out.println("<input type=\"submit\" value=\"Ripristina\">");
        out.println("</form></td><td>");
        out.println("<form action=\"windows\" method=\"post\">");
        out.println("<input type=\"hidden\" name=\"windowName\" value=\"\"+name+\"\">");
        out.println("<input type=\"hidden\" name=\"action\" value=\"close\">");
        out.println("<input type=\"submit\" value=\"Chiudi\">");
        out.println("</form></td></tr></table></td></tr>");
        if (!window.isIconified( )) {
            out.println("<tr><td height=\"\"+window.getY( )+\"\" align=\"\"+align+\"\"><br><br>");
            out.println("<font color=\"\"+color+\"\">");
            return EVAL_BODY_INCLUDE;
        } // fine if
    } catch(IOException e) {return SKIP_BODY;}
    return SKIP_BODY;
}
public int doEndTag( ) {
    if (window.isClosed( )) return EVAL_PAGE;
    try {
        JspWriter out = pageContext.getOut();
        if (!window.isIconified( )) {
            out.println("</font>");
            out.println("<br><br></td></tr><tr>");
        }
    }
}

```

```

out.println("<td align=\"right\" bgcolor=\""+borderColor+"\" height=\"10\">");
out.println("<form action=\"windows\" method=\"post\">");
out.println("<table><tr><td>");
out.println("X <input type=\"text\" name=\"x\" value=\""+window.getX()+"\" size=\"3\">");
out.println("</td><td>");
out.println("Y <input type=\"text\" name=\"y\" value=\""+window.getY()+"\" size=\"3\">");
out.println("</td><td>");
out.println("<input type=\"hidden\" name=\"windowName\" value=\""+name+"\">");
out.println("<input type=\"hidden\" name=\"action\" value=\"resize\">");
out.println("<input type=\"submit\" value=\"Ridimensiona\">");
out.println("</td></tr></table></form></td></tr>");
}
out.println("</table>");
} catch(IOException e) {}
return EVAL_PAGE;
}
}
}

```

Esempio 1.5 Il file WindowTag.java

Durante l'esecuzione della pagina JSP, non appena viene incontrato il tag esteso window la classe sopra scritta, che rappresenta il tag stesso, viene caricata e le proprietà (name, bgColor, align, etc...) sono automaticamente riempite col valore del rispettivo attributo del tag. Vengono chiamati i metodi doStartTag() e doEndTag() che non fanno altro che generare come output una tabella HTML al cui interno è contenuto il corpo del tag esteso. La tabella è la stessa dell'esempio precedente, infatti i due metodi accedono alle proprietà dell'oggetto window messo in sessione; è possibile che esistano più finestre e quindi l'attributo name del tag esteso è usato per individuare l'oggetto window, e se questo non esiste viene messo in sessione usando proprio tale nome.

Quando si ricorre ai tag estesi occorre anche definire un file descrittore che prende il nome di **Tag Library Descriptor File** (tld). Questo file serve per elencare tutti i tag della libreria, le relative classi Java a cui corrisponde, e i vari attributi che possono essere necessari o no.

```

<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE taglib
PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.1//EN"
"http://java.sun.com/j2ee/dtds/web-jsptaglibrary_1_1.dtd">
<taglib>
  <tlibversion>1.0</tlibversion>
  <jspversion>1.1</jspversion>
  <shortname>ex</shortname>
  <info>Un esempio di tag library creato da Fabio Rombaldoni</info>
  <tag>
    <name>window</name>
    <tagclass>WindowTag</tagclass>
    <bodycontent>JSP</bodycontent>
    <info>Tag usato per disegnare una finestra</info>
    <attribute><name>name</name><required>true</required></attribute>
    <attribute><name>color</name><required>>false</required></attribute>
    <attribute><name>bgColor</name><required>>false</required></attribute>
    <attribute><name>border</name><required>>false</required></attribute>
    <attribute><name>align</name><required>>false</required></attribute>
    <attribute><name>width</name><required>>false</required></attribute>
    <attribute><name>height</name><required>>false</required></attribute>
    <attribute><name>borderColor</name><required>>false</required></attribute>
  </tag>
</taglib>

```

Esempio 1.6 Il Tag Library Descriptor file per l'esempio

Adesso si può finalmente tornare alla pagina JSP e vedere come essa realmente deve essere scritta.

```
<%@ page language="java" contentType="text/html" %>
<%@ taglib uri="ex-taglib.tld" prefix="ex" %>
<html>
<head>
  <title>MVC Pattern - Tag Library</title>
</head>
<body>
<ex:window name="Window1" border="10" color="white" bgColor="blue" align="center" width="100"
height="100" borderColor="red">
Questo è il contenuto della finestra!!!
</ex:window>
<br><br>
<ex:window name="Window2" border="5" color="red" bgColor="yellow" align="right" width="100"
height="100" borderColor="green">
Questa è un'altra finestra ancora!!!
</ex:window>
</body>
</html>
```

Esempio 1.7 Il file windows.jsp migliorato

L'istruzione `<%@ taglib ...%>` serve per individuare la posizione del tld e per identificare la libreria dei tag con un prefisso 'ex'; questo meccanismo permette di evitare conflitti con tag omonimi ma appartenenti a librerie diverse. Grazie alle librerie di tag è possibile estendere i tag di base del linguaggio JSP e creare tag ad hoc per lo scopo che si deve raggiungere.

Ancora una volta l'esempio fin qui mostrato non è immune da possibili miglioramenti. Si pensi alla possibilità di generare tanti stili diversi della solita finestra anche a seconda del tipo di client: browser Web o cellulare Wap.

Capitolo 4 Il Client Tier

Appartengono allo strato client (Client Tier) le applicazioni che forniscono all'utente una interfaccia semplificata verso il mondo enterprise, e che quindi rappresentano la percezione che l'utente ha dell'applicazione J2EE. Tali applicazioni si suddividono in due classi di appartenenza: le applicazioni Web-based e le applicazioni non Web-based.

Le prime sono quelle applicazioni che utilizzano il browser Web (es. Netscape Navigator) come strato di supporto all'interfaccia verso l'utente, ed i cui contenuti sono generati dinamicamente da servlet o pagine JSP o staticamente in HTML. Le seconde sono invece applicazioni stand-alone che sfruttano lo strato di rete disponibile sul client per interfacciarsi direttamente con l'applicazione J2EE, senza passare per il Web Tier.

Nella Figura 4.1 sono illustrate schematicamente le applicazioni appartenenti a questo strato e le modalità di interconnessione verso gli strati che compongono l'architettura del sistema.

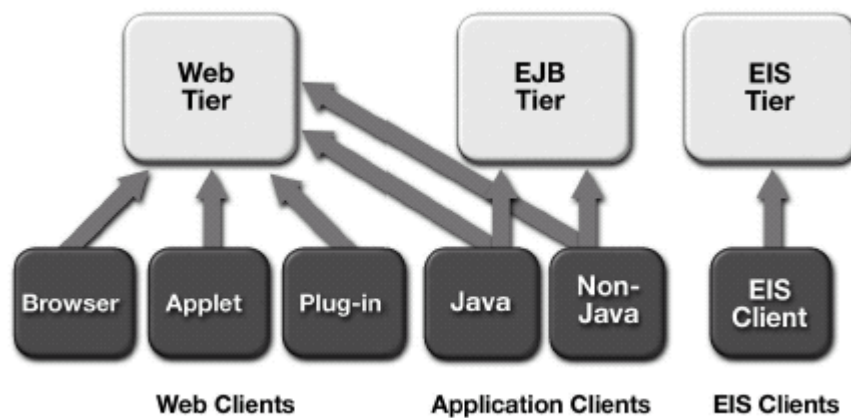


Figura 4.1 Client J2EE

Molti aspetti del client sono determinati dal tier dell'applicazione enterprise cui il client si connette. Questa discussione classifica i client in tre ampie categorie, basate sui tier con i quali interagiscono:

- ✚ I **client Web** si connettono al Web Tier. Eseguono su un desktop o su altro host browser all'interno di un browser Web o su un browser plug-in. Sia la presentation logic sia la business logic dell'applicazione possono girare o sul server o sul client.
- ✚ I **client EJB** si connettono all'EJB Tier. Sono tipicamente programmi GUI che eseguono su un desktop. Hanno accesso a tutte le facility fornite dall'EJB Tier della piattaforma J2EE. La presentation logic dell'applicazione gira sul client, mentre la business logic gira su server.
- ✚ I **client EIS** si interfacciano direttamente con le risorse di un EIS. Tipicamente sono programmi utili all'amministrazione, al controllo e alla gestione dei sistemi di back-end. Sia la presentation sia la business logic sono contenute nel client.

In generale un'applicazione J2EE dovrebbe usare client Web quando possibile. Con l'apporto di tecnologie come DHTML (Dyanamic HTML) e JavaScript, i browser Web possono supportare interfacce utente ragionevolmente potenti e veloci.

4.1 I Client Web

I client Web solitamente girano all'interno di un browser e usano i servizi messi a disposizione dal browser per visualizzare i contenuti forniti dal Web Tier. In questi client, l'interfaccia utente è generata lato server dal Web Tier e comunicata via HTML. Applet e JavaScript possono essere usati per migliorare l'interfaccia di navigazione. Sul lato client, un browser, od un programma equivalente, visualizzano questa interfaccia e permettono le interazioni con l'utente. Il data interchange è facilitato dall'XML.

I client Web usano i protocolli **HTTP** o **HTTPS**, i quali hanno diversi vantaggi. Infatti questi protocolli sono largamente deployati, sono robusti e semplici, e passano attraverso i firewall.

Parallelamente, la semplicità di tali protocolli presenta due svantaggi che possono, in ogni caso, essere superati. Primo, sono **sessionless**; HTTP è un protocollo request/response, con nessun concetto di sessione incorporato. Secondo, sono **non-transazionali**; HTTP è un protocollo di rete, non progettato per il Distributed Computing. Conseguenza di ciò, non ha alcun concetto di transazione o di security context. Comunque, questo non è un problema in pratica, dato che le transazioni e la sicurezza sono comunemente gestite sul server.

I contenuti trasmessi sopra il protocollo HTTP sono generalmente il risultato di una azione realizzata sul server in risposta ad una richiesta del client. Questo risultato può essere formattato usando HTML o XML.

Molti client Web girano all'interno o in unione con un browser Web. Il browser può gestire i dettagli della comunicazione HTTP e del rendering HTML, mentre un componente dell'applicazione può focalizzarsi sulle interazioni che non possono essere tradotte in HTML. Un browser Web senza altri componenti è il più semplice e diffuso client Web J2EE. Componenti aggiuntivi come applet Java, browser plug-in nel browser Netscape, o componenti ActiveX nel browser Internet Explorer di Microsoft, possono rendere l'interfaccia utente più ricca e caratteristica. In ultimo, i client Web possono essere implementati come programmi stand-alone, nella forma di client Java la cui interfaccia utente è tipicamente scritta usando l'API Swing o di client non-Java scritti in C++ o in linguaggi di automazione come Visual Basic.

Un client Web consiste di due parti: pagine Web dinamiche contenenti vari tipi di linguaggi di markup (come HTML e XML), generate dai componenti Web che girano sul Web Tier, ed un browser Web, che mostra le pagine ricevute dal server.

A volte un client Web è detto **thin client**. I thin client solitamente non fanno cose come interrogare un database, realizzare complesse business rule, o connettersi ad applicazioni legacy. Quando si usa un thin client, operazioni "pesanti" come le precedenti sono a carico di enterprise bean che eseguono sul J2EE server, dove possono sfruttare la sicurezza, la velocità, i servizi e l'affidabilità delle tecnologie J2EE server-side.

4.2 I Client EJB

I client EJB sono application client che interagiscono con l'EJB Tier. Sono programmi GUI che tipicamente eseguono su un desktop; essi gestiscono la loro GUI offrendo una user experience simile a quella delle applicazioni native.

I client EJB possono essere implementati sia in Java sia in linguaggi come C++ o Visual Basic. Quando viene usato un linguaggio diverso da Java, dipendentemente dall'implementazione del container, alcune facility della piattaforma J2EE potrebbero non essere disponibili.

4.3 Design per molteplici tipi di Client

Questa sezione tratta gli approcci alla progettazione di applicazioni enterprise che possano supportare molteplici tipi di client. I dati dell'applicazione e le business rule sono indipendenti dal client che accede all'applicazione, e ciò rende desiderabile che questi oggetti siano progettati per essere condivisi dai diversi client dell'applicazione.

Quando diversi tipi di client presentano la stessa funzionalità attraverso interfacce differenti, è utile condividere oggetti che incapsulano questa funzionalità. La distinzione tra oggetti che possono essere condivisi o riusati, ed oggetti che devono essere implementati separatamente per ciascun tipo di client può essere discussa in termini di architettura MVC.

Il Model è un'astrazione software dei dati dell'applicazione e delle business rule che possono modificare questi dati e può essere condiviso tra tutti i client dell'applicazione. Il Model dovrebbe essere coerente indifferentemente dal tipo di client che accede ai dati. Se il Model riproduce fedelmente tutti i possibili modi con cui i dati possono essere cambiati, non c'è alcuna necessità di implementare classi Model differenti, o sviluppare oggetti Model specifici per ogni tipo di client.

Una View visualizza i contenuti di un Model. Essa accede ai dati del Model e specifica come i dati dovrebbero essere presentati. La View cambia significativamente a seconda del client. Ciò rende difficile condividere implementazioni complete della View.

Un Controller definisce il comportamento dell'applicazione; esso interpreta gli eventi utente e li associa ad azioni che devono essere realizzate dal Model. Ciascun client che mostra funzionalità differenti richiede un Controller separato.

4.4 Considerazioni ulteriori sui Client J2EE

La piattaforma J2EE supporta diversi tipi di client. Molti servizi sono disegnati per supportare come client i browser Web. Tali servizi interagiscono con i loro client tramite form e pagine HTML generate dinamicamente. Servizi più sofisticati interagiscono con i loro client del first-tier direttamente scambiandosi dati aziendali. In questo caso, JSP e servlet sono usati per formattare questi dati in un modo con cui è facile lavorare per i client J2EE. Tali client possono essere sia applet Java in esecuzione in un browser Web sia programmi basati su tecnologia Java.

4.4.1 Client basati su pagine HTML

Un servizio può essere presentato direttamente ad un browser Web utente come pagina HTML generata dinamicamente. La tecnologia JSP fornisce un modo facile per comporre tali pagine usando un paradigma di scripting familiare che combina HTML e codice Java. In alcuni casi, un servizio potrebbe richiedere codice discretamente complesso. Tale complessità può essere gestita ponendo il codice in un componente JavaBeans e richiamando il bean da una pagina JSP. Un servizio può anche essere direttamente programmato in linguaggio Java usando un servlet.

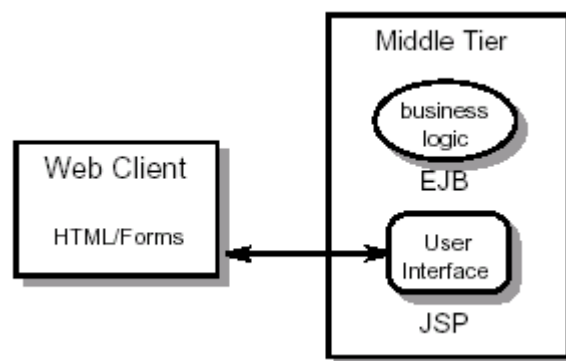


Figura 4.2 Presentare servizi direttamente ad un browser

4.4.2 Client HTTP Content-Based

Spesso può risultare utile fornire direttamente al client funzionalità che aiutino un utente ad organizzare ed interagire con le informazioni del servizio. Per implementare questo tipo di servizio il contenuto è tipicamente nella forma di documenti XML scambiati tra client e servizio usando il protocollo HTTP. Generalmente questo contenuto XML è gestito nel first-tier da componenti JavaBeans forniti dal servizio in un applet scaricata automaticamente nel browser dell'utente, come mostrato in Figura 4.3 nella pagina seguente. Per evitare i problemi causati da versioni vecchie e non-standard dell'ambiente runtime Java in un browser, il modello J2EE fornisce supporto speciale per scaricare ed installare automaticamente i Java Plug-in, cioè ambienti runtime Java di Sun che possono essere caricati dinamicamente nei più popolari browser Web.

Questo contenuto può essere gestito anche da un programma Java puro. Questo modello client flessibile fornisce allo sviluppatore un'ampia scelta per presentare l'interfaccia utente di una applicazione distribuita su Internet.

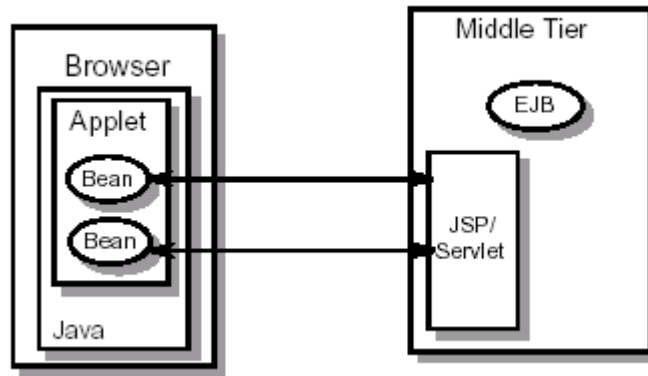


Figura 4.3 Fornire componenti JavaBeans ad un browser

4.4.3 Client Intranet

Servizi basati su pagine HTML e servizi basati su HTTP content possono essere efficacemente usati sia su una Intranet aziendale sia su Internet. In aggiunta, l'Intranet fornisce l'infrastruttura supplementare che permette ai programmi Java di accedere direttamente agli EJBs all'interno del dominio Intranet.

4.4.4 Altri tipi di client

I servizi presentati tramite gli standard HTTP, HTML e XML sono facilmente accessibili da tutti i client, inclusi quelli Microsoft come Visual Basic e Office 2000.

Uno degli scopi della tecnologia EJB è di definire RMI-IIOP quale meccanismo richiesto di interoperabilità. Ciò renderà ciascun servizio J2EE disponibile a ciascun client CORBA, assicurando un'integrazione più completa tra la piattaforma J2EE e gli EISs esistenti.

La piattaforma J2EE fornisce tecniche per integrare oggetti COM Microsoft con gli EJBs usando RMI-IIOP. Questo permette a client quali Visual Basic e Office 2000 di accedere agli EJBs via COM, così come usare EJBs in combinazione con funzioni middle-tier implementate in Microsoft Transaction Server.

Capitolo 5 Il Web Tier

A partire dai primi siti internet dai contenuti statici, oggi siamo in grado di poter effettuare qualsiasi operazione tramite Web da remoto. Questa enorme crescita, come una vera rivoluzione, ha voluto le sue vittime. Nel corso degli anni sono nate e poi completamente scomparse un gran numero di tecnologie a supporto di un sistema in continua evoluzione.

Gli utenti hanno beneficiato significativamente dalla aumentata capacità di applicazioni Web di generare contenuti dinamici customizzati sui loro bisogni. JavaServer Pages e Servlet sono le tecnologie J2EE che supportano la generazione di contenuti dinamici in maniera portabile e cross-platform. Le applicazioni J2EE Web-based che usano tali tecnologie possono essere progettate in diversi modi. Applicazioni semplici possono usare pagine base JSP e servlet o pagine JSP con componenti modulari. Applicazioni J2EE transazionali più complesse usano pagine JSP e componenti modulari con enterprise bean.

Il **Web server** gioca un ruolo fondamentale all'interno del Web Tier. In ascolto su una macchina server, riceve richieste da parte del browser (client), le processa, quindi restituisce al client una entità o un eventuale codice di errore come prodotto della richiesta.

Il Netcraft Web Server Survey è una stima dell'uso e della distribuzione dei software Web server sui computer connessi ad Internet. Qui sotto sono riportati i dati relativi alla distribuzione dei Web server sul mercato negli ultimi anni aggiornati a Marzo 2002 (i dati sono disponibili nella forma completa all'indirizzo Internet <http://www.netcraft.com/survey>).

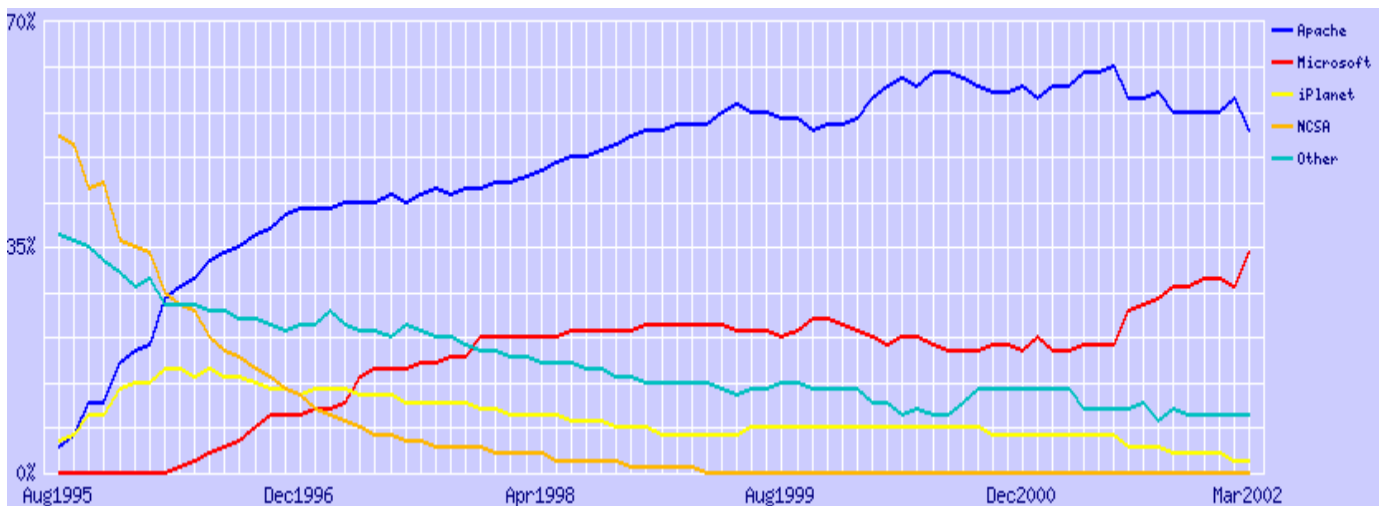


Grafico 5.1 Market Share for Top Servers Across All Domains August 1995 - March 2002

Developer	February 2002	Percent	March 2002	Percent	Change %
Apache	22.462.777	58,43	20.492.088	53,76	-4,67
Microsoft	11.198.727	29,13	12.968.860	34,02	4,89
iPlanet	1.123.701	2,92	889.857	2,33	-0,59
Zeus	837.968	2,18	855.103	2,24	0,06

Tabella 5.1 Top Developers

Developer	February 2002	Percent	March 2002	Percent	Change %
Apache	10.147.402	65,18	9.522.954	64,37	-0,81
Microsoft	4.069.193	26,14	3.966.743	26,81	0,67
iPlanet	283.112	1,82	265.826	1,8	-0,02
Zeus	177.225	1,14	170.023	1,15	0,01

Tabella 5.2 Active Sites

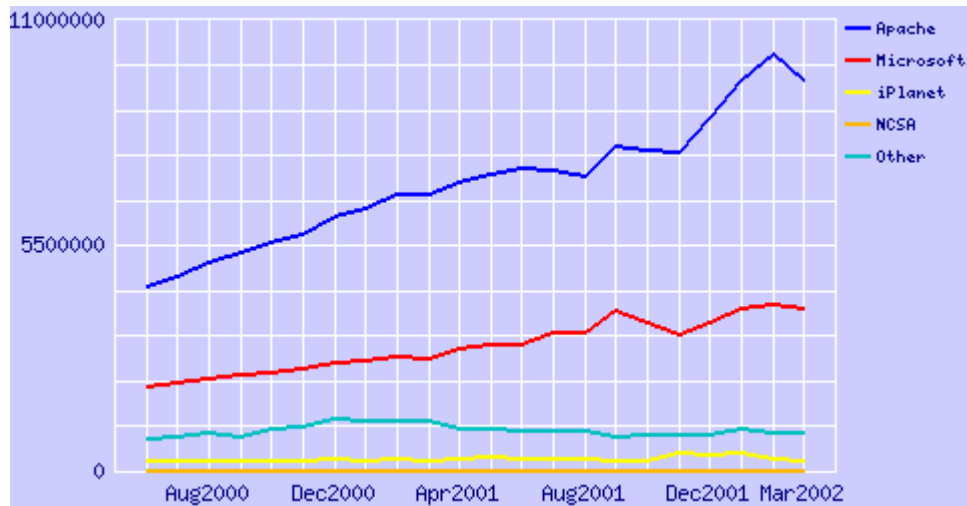


Grafico 5.2 Totals for Active Servers Across All Domains June 2000 - March 2002

Nei grafici e tabelle mostrati precedentemente, **iPlanet** comprende iPlanet-Enterprise, Netscape-Enterprise, Netscape-FastTrack, Netscape-Commerce, Netscape-Communications, Netsite-Commerce & Netsite-Communications, mentre **Microsoft** comprende Microsoft-IIS, Microsoft-IIS-W, Microsoft-PWS-95, & Microsoft-PWS.

5.1 Applicazioni Web e Web container

Un'**applicazione Web** è un insieme di documenti HTML/XML, componenti Web (servlet e pagine JSP) ed altre risorse presenti in una struttura a directory o archiviate in un formato conosciuto come Web Archive (WAR) file. Un'applicazione Web risiede su un server centrale e fornisce servizi ad una moltitudine di client. Le applicazioni Web forniscono contenuti dinamici ed interattivi a client browser-based. Applicazioni Web browser-based possono essere usate per qualsiasi tipo di applicazione: da applicazioni di sicurezza B2B a siti Web per il commercio elettronico.

Un **Web container** è un ambiente runtime per un'applicazione Web: questa gira all'interno di un Web container di un Web server. Un Web container fornisce ai componenti un contesto di naming e gestione del ciclo di vita. Pertanto, come un client appartenente allo strato Client Tier rappresenta la percezione che l'utente ha dell'applicazione J2EE, il contenitore rappresenta la percezione che un componente al suo interno ha della interazione verso l'esterno (Figura 5.1).

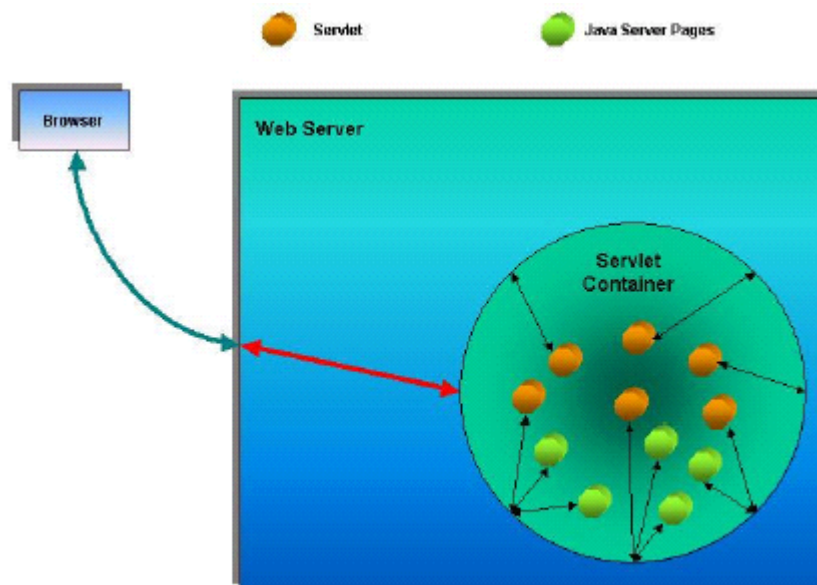


Figura 5.1 Applicazioni Web e Web container

5.2 Sviluppo di applicazioni Web e Creazione di contenuti dinamici

Esistono diverse tecnologie per sviluppare applicazioni Web e per la creazione di contenuti dinamici, che vanno dalla convenzionale CGI, a soluzioni proprietarie come ISAPI di Microsoft e NSAPI di Netscape, alla tecnologia Active Server Pages (ASP) di Microsoft, fino alle tecnologie Java Servlet e JavaServer Pages (JSP) proposte da Sun.

5.2.1 Common Gateway Interface (CGI)

Sebbene Internet fu originariamente progettato per fornire contenuti statici, la necessità di presentare contenuti dinamici, disegnati sui bisogni dell'utente, ha rapidamente guidato lo sviluppo di tecnologie Web. La prima risposta a questo bisogno fu la tecnologia CGI (Common Gateway Interface), uno standard che consente di far comunicare le richieste HTTP con una applicazione residente sul server. Questa interfaccia permette ai Web server di ottenere/inviare dati da/a database, documenti o altri programmi, e presentare tali dati all'utente attraverso il Web. CGI è quindi la più comune e datata tra le tecnologie server-side per lo sviluppo di applicazioni Web. Definisce come il client effettua il passaggio dei dati di input e la chiamata al programma sul server e come il server restituisce i dati di output del programma al client. CGI deve funzionare all'interno del protocollo HTTP, il quale è usato come supporto in entrambe le direzioni. L'interazione tra browser e applicazione CGI segue queste fasi:

- 1) **Acquisizione dei dati (lato Client):** i dati sono acquisiti dal lato client mediante form. Il tag HTML per il data input è `<form></form>`, e gli attributi tipici di tale tag sono **action**=URL e **method**="GET"|"POST". Il form consente di raccogliere dati di diversi tipi e con diversi stili di interazione. Tali dati devono essere mantenuti in associazioni nome=valore; ogni elemento del form deve quindi avere un nome. Il programma che riceverà i dati da elaborare, ricostruirà le strutture dati a partire dalle associazioni nome=valore. Esistono diverse tipologie di input: text, hidden, password, checkbox, radiobutton, textarea, select/option.
- 2) **Invocazione del programma server-side e spedizione dei dati (lato Client):** il browser invia al server Web una richiesta facendo riferimento al nome dell'applicazione seguita da una serie di parametri; per inviare il contenuto del form ad un programma residente su server si deve clickare sul bottone di **submit** (es. `<input type=submit value="Click me">`). All'evento di submit è associata l'invocazione del programma server-side specificato nell'attributo action. I dati sono spediti al server in coppie nomevariabile=valore separate da un simbolo &.
- 3) **Lancio del programma (lato Server):** il server Web riconosce che la richiesta deve essere indirizzata all'applicazione specificata e ne attiva un'istanza passandole opportunamente i parametri; dal lato server il programma per partire deve essere ospitato in un ambiente all'interno del quale il server definisce delle speciali variabili, chiamate appunto **variabili d'ambiente**, il cui valore viene ricavato dalle informazioni inviate dal browser. Le variabili d'ambiente danno informazioni sul client, sul server e sull'interazione tra client e server.

Le variabili d'ambiente più importanti che descrivono il server sono:

- ❑ SERVER_SOFTWARE: il nome e la versione del server HTTP che risponde alla richiesta CGI;
- ❑ SERVER_NAME: il nome del server (può essere un alias di DNS o un indirizzo IP numerico);
- ❑ SERVER_PORT: il numero di porta cui è indirizzata la richiesta (porta 80, generalmente).

Le variabili d'ambiente più importanti che descrivono il client sono:

- ❑ REMOTE_HOST: il nome dell'host che invia la richiesta;
- ❑ REMOTE_ADDR: l'indirizzo IP dell'host che invia la richiesta;
- ❑ AUTH_TYPE: il metodo usato dal protocollo di autenticazione dell'utente;
- ❑ REMOTE_USER: il nome dell'utente autenticato;
- ❑ HTTP_ACCEPT: i tipi MIME accettati dal client;
- ❑ HTTP_USER_AGENT: il nome e la versione del browser che ha inviato la richiesta.

Le variabili più significative sono quelle che descrivono l'interazione tra client e server:

- ❑ GATEWAY_INTERFACE: la versione delle specifiche CGI;
- ❑ SERVER_PROTOCOL: il nome e la versione del protocollo di comunicazione HTTP;

- ❑ SCRIPT_NAME: il path completo del programma CGI;
- ❑ REQUEST_METHOD: il metodo usato per la richiesta (GET, POST, ed altri);
- ❑ QUERY_STRING: contiene i dati passati dalle richieste GET, ISINDEX, ISMAP;
- ❑ CONTENT_TYPE: specifica delle richieste POST e PUT, questa variabile contiene il tipo MIME dei dati passati al CGI;
- ❑ CONTENT_LENGTH: la lunghezza dei dati passati con il metodo POST.

- 4) **Elaborazione e decodifica dei dati e produzione dell'output (lato Server)**: l'applicazione CGI effettua l'elaborazione in base ai parametri acquisiti e crea sullo standard output un flusso di dati che costituisce la pagina di risposta per il browser; sulla base delle variabili di ambiente il programma sa come decodificare i dati in arrivo. La decodifica dipende innanzi tutto dal metodo usato per il trasferimento (attributo method del tag form), che può essere GET o POST. Nel caso di metodo **GET**, il programma per ottenere i valori in input deve leggere il valore della variabile QUERY_STRING, che contiene le coppie nomevariabile=valore separate da &, e decodificare i dati. Nel caso di metodo **POST**, le coppie nomevariabile=valore separate da & sono inserite nello standard input del programma; QUERY_STRING non contiene nulla mentre CONTENT_TYPE e CONTENT_LENGTH contengono rispettivamente il tipo MIME dei dati passati e la loro lunghezza. La differenza primaria tra GET e POST sta dunque nel passaggio dei valori. GET è il metodo di default e va bene in tutte le query che non richiedono "data encoding". Comunque, in entrambi i casi l'input del programma risulta composto da una lunga sequenza di caratteri in cui vanno selezionati i valori nelle coppie nomevariabile=valore e assegnati ad opportune variabili interne del programma e vanno sostituiti i caratteri protetti (es. + va sostituito dallo spazio). L'output del programma verrà reinviato al client e perché ciò sia possibile deve essere in un formato compatibile con le CGI. Le CGI reindirizzano direttamente lo standard output del programma al client. Tipicamente il formato più adatto per avere un output formattato è l'HTML. La prima riga da stampare sullo standard output deve indicare il tipo MIME dell'output vero e proprio che seguirà (es. echo 'Content-type: text/html'). Questa riga non viene inviata al client ma usata dal server HTTP per impostare la comunicazione con il browser. Le successive "stampe" devono essere conformi al tipo MIME specificato nella prima riga (es. echo '<HTML><BODY><H1>ciao!</H1></BODY></HTML>').
- 5) **Reindirizzamento dell'output al client (lato Server)**: il server Web cattura il flusso di dati generato dall'applicazione CGI e la trasforma in una risposta HTTP inviata al client, il quale visualizzerà l'output.

L'applicazione CGI può essere uno script (ovvero un programma interpretato) di shell del sistema operativo o di un linguaggio interpretato (ad esempio Perl), oppure un'applicazione vera e propria sviluppata con un qualsiasi linguaggio di programmazione e che segue alcune regole per interfacciarsi con il Web (tipicamente programmi compilati C, C++, Fortran, Prolog).

```
#!/bin/sh
# ciao.sh.cgi
echo "Content-type: text/html"
echo
# html header
echo "<HTML><HEAD><TITLE>Ciao!</TITLE></HEAD>"
# html body
echo "<BODY><H1>CIAO MONDO!</H1></BODY></HTML>"
```

Esempio 5.1 CIAO MONDO! in Shell

```
#!/usr/bin/perl
# ciao.perl.cgi
print "Content-type: text/html\n\n";
# html header
print "<HTML><HEAD><TITLE>ciao!</TITLE></HEAD>\n";
# html body
print "<BODY><H1>CIAO MONDO!</H1></BODY></HTML>\n";
```

Esempio 5.2 CIAO MONDO! in PERL

```

<html>
<head>
  <title>Prova FORM e CGI </title>
</head>
<body>
  <p>
    <b>Prova FORM e CGI</b>
    <form action="prova.perl.cgi" method="get" >
      nome <input type="text" name="nome" size="20"><br>
      cognome <input type="text" name="cognome" size="20"><br>
      <input type="submit" value="Click me">
    </form>
  </p>
</body>
</html>

```

Esempio 5.3 Form HTML per richiesta nome/cognome

```

#!/usr/bin/perl
# prova.perl.cgi
print "Content-type: text/plain\n\n";
print "prova form nome e cognome\n\n";
# salva il valore della QUERY_STRING in una variabile qs
$qs = $ENV{'QUERY_STRING'};
# la divide in un array nomevar=valore sulla base degli &
@qs = split(/&/,$qs);
...
foreach $i (0 .. $#qs) {
  # converte i + in spazi
  $qs[$i] =~ s/\+/ /g;
  # converte i caratteri esadecimali
  $qs =~ s/%(..)/pack("c",hex($1))/ge;
  # divide ogni variabile in nomevar e valore
  ($name, $value) = split(/=/,$qs[$i],2);
  # crea un array con indice il nome e valore il valore
  $qs{$name} = $value;
}
print "\variabili:\n\n";
foreach $name (sort keys(%qs))
{ print "$name=", $qs{$name}, "\n" }

```

Esempio 5.4 Programma PERL richiamato dalla pagina HTML precedente

Alcuni tra gli esempi appena presentati sono scritti in PERL (acronimo di Processing Extraction Report Language). **PERL** è un linguaggio di scripting pensato per fare amministrazione di sistema con script batch. E' molto adatto per gli script CGI perché è simile al C ma più semplice, ha primitive che rendono molto agile operare con le stringhe ed interagisce bene (ovviamente) con il sistema operativo.

PERL è un linguaggio interpretato, quindi va eseguito insieme al suo interprete. Per questo motivo i file PERL iniziano tutti con il path (relativo) dell'interprete, tipicamente **#!/usr/bin/perl**. Tutti i nomi delle variabili sono case-sensitive e iniziano con il simbolo \$.

La tecnologia CGI presenta comunque diversi limiti.

Primo, il codice entro uno script CGI che accede a risorse, quali database o file system, deve essere **specifico della piattaforma server**. Dunque, molte applicazioni CGI non potranno girare su altre piattaforme server senza alcuna modifica. Questo limita decisamente il loro utilizzo in un ambiente distribuito dove le applicazioni Web devono girare su piattaforme molteplici.

Il secondo e più grande svantaggio nell'uso dei CGI sta nella **scarsa scalabilità** della tecnologia: infatti, ogni volta che il Web server riceve una richiesta deve creare una nuova istanza del CGI. Il dover creare un processo per ogni richiesta ha come pessima conseguenza un impiego intensivo

delle risorse ed enormi carichi in fatto di tempi d'esecuzione: i CGI sono quindi lenti e tendono a scalare non troppo bene. L'aumento di risorse HW permette di scalare relativamente. Il grado con il quale l'applicazione scalerà è comunque limitato dall'hardware e dal sistema operativo. A meno di non utilizzare strumenti di altro tipo, è inoltre impossibile riuscire ad ottimizzare gli accessi a database. Ogni istanza del CGI sarà infatti costretta ad aprire e chiudere una propria connessione verso il DBMS. Questi problemi sono stati affrontati ed in parte risolti dai **Fast-CGI** grazie ai quali è possibile condividere una stessa istanza tra più richieste HTTP.

Terzo, le CGI **non hanno la possibilità di mantenere lo stato di una sessione**; è quindi compito del programmatore portare per ogni richiesta e risposta lo stato dell'utente.

Per ultimo, le applicazioni CGI sono **difficili da mantenere** perché combinano la logica relativa ai contenuti con quella relativa alla presentazione degli stessi in un unico codice di base. Come conseguenza, sono necessarie due tipi di competenza per mantenere ed aggiornare script CGI.

Molti produttori di Web server hanno migliorato CGI per il loro specifico prodotto ed hanno sviluppato modi migliori di gestire funzioni CGI-like fornendo estensioni ai loro prodotti. Questo ha permesso lo sviluppo di sofisticate applicazioni basate su CGI. Comunque i problemi radicati in tale tecnologia ancora esistono: le applicazioni CGI sono platform-specific, non scalano bene e sono difficili da mantenere.

5.2.2 ISAPI ed NSAPI

Per far fronte ai limiti tecnologici imposti dai CGI, Microsoft e Netscape hanno sviluppato API proprietarie (rispettivamente Internet Server API e Netscape Server API) mediante le quali creare librerie ed applicazioni che possono essere caricate dal Web server al suo avvio ed utilizzate come proprie **estensioni**.

Considerando **ISAPI**, è un'interfaccia per permettere agli sviluppatori di creare delle DLL che funzionano con il server Web IIS. Una **Dynamic-Link Library** è un file contenente un insieme di funzioni riutilizzabili; queste librerie si possono collegare nello spazio di memoria di un eseguibile per fornire funzionalità aggiuntive in fase d'esecuzione. Dato che le applicazioni ISAPI sono strutturalmente delle DLL, vengono caricate in memoria e inicializzate solo una volta. Chiamate successive di un browser ad una DLL richiedono meno lavoro aggiuntivo perché la libreria è già presente in memoria. In aggiunta, poiché la DLL è a conoscenza di eventuali chiamate precedenti, mantiene informazioni di stato per l'intera sessione con il browser. Una particolare applicazione ISAPI è **Internet Database Connector (IDC)**, che realizza un meccanismo per accedere ad una qualsiasi sorgente di dati **ODBC** (Open DataBase Connectivity).

L'insuccesso di queste tecnologie (ISAPI e NSAPI) è stato però segnato proprio dalla loro natura di **tecnologie proprietarie** e quindi non portabili tra le varie piattaforme sul mercato. Oltre a ciò, essendo utilizzate come moduli del Web server è caso frequente che un loro accesso errato alla memoria causi il crash dell'intero Web server provocando enormi danni al sistema, che deve ogni volta essere riavviato.

5.2.3 Active Server Pages (ASP)

Active Server Pages è stata l'ultima tecnologia Web rilasciata da **Microsoft**. Introdotta con il server Web **Internet Information Server 3.0**, è una particolare applicazione ISAPI e consente lo sviluppo di pagine HTML create dinamicamente, in grado di supportare scripting sul lato server e capaci di interagire con componenti ed applicazioni in esecuzione sempre sul server.

Un'applicazione ASP è tipicamente un mix tra HTML e linguaggi script come VBScript, Jscript o JavaScript, mediante i quali si può accedere ad oggetti del server; gli script server-side elaborano le richieste provenienti dai browser client e possono creare una pagina di risposta per ciascun client particolare, interrogando ad esempio un database tramite Active Data Objects (ADO).

Questa capacità è molto importante, perché consente di creare pagine HTML dinamiche che possono essere scaricate da qualsiasi browser che supporta HTML semplice. Una pagina ASP, a differenza di ISAPI, non viene eseguita come estensione del Web server, ma viene compilata alla prima chiamata, ed il codice compilato può quindi essere utilizzato ad ogni richiesta HTTP.

L'aspetto più interessante di ASP consiste, quindi, nel produrre codice HTML sul server e successivamente inviarlo al browser per la sua visualizzazione. In pratica, parte della logica elaborativa necessaria per realizzare pagine Web interattive e dinamiche viene spostata dal client verso il server con evidenti vantaggi.

Nonostante sia oggi largamente diffusa, ASP come ISAPI rimane una tecnologia proprietaria e quindi in grado di funzionare solamente su piattaforma Microsoft; ma il reale svantaggio di ASP è però legato alla sua natura di mix tra linguaggi script ed HTML. Questa sua caratteristica ha come effetto secondario quello di riunire in un unico contenitore sia le logiche applicative sia le logiche di presentazione rendendo estremamente complicate le normali operazioni di manutenzione delle pagine.

5.2.4 Breve nota su JavaScript

JavaScript nasce come un linguaggio di scripting (=interpretato), client side (=eseguito dal browser lato client), application embedded (=codice incluso nell'HTML usando il tag <script>). È stato poi esteso per essere eseguito server-side. Le istruzioni client-side contengono il supporto per accedere all'ambiente applicativo (il browser) e al suo contenuto (la pagina HTML). Le istruzioni server-side contengono supporto per connettersi ai principali DBMS, per accedere al file system del server e per comunicare con altre applicazioni.

5.2.5 Introduzione alle tecnologie Sun: Servlet e JavaServer Pages

Sun cerca di risolvere i problemi legati alle varie tecnologie illustrate, mettendo a disposizione due tecnologie che, raccogliendo l'eredità dei predecessori, ne abbattano i limiti, mantenendo e migliorando gli aspetti positivi di ognuna. In aggiunta, Servlet e JavaServer Pages ereditano tutte quelle caratteristiche che rendono le applicazioni Java potenti, flessibili e semplici da mantenere: portabilità del codice e paradigma object oriented.

Rispetto ai CGI, i servlet forniscono un ambiente ideale per quelle applicazioni per cui sia richiesta una massiccia scalabilità e di conseguenza l'ottimizzazione degli accessi alle basi dati di sistema. Rispetto ad ISAPI ed NSAPI, pur rappresentando un'estensione al Web server, mediante il meccanismo delle eccezioni risolvono a priori tutti gli errori che potrebbero causare la terminazione prematura del sistema. Rispetto ad ASP, Servlet e JSP separano completamente le logiche di business dalle logiche di presentazione, dotando il programmatore di un ambiente semplice da modificare o mantenere. La piattaforma J2EE supporta queste due tecnologie.

5.2.5.1 La tecnologia Servlet

I servlet sono moduli scritti in Java utilizzando la Java Servlet API, diventata un'estensione standard di Java dalla versione 1.2, che permettono di estendere le funzionalità di un Web server. Ad esempio, un servlet può prelevare dati da un form HTML, aggiornare le informazioni di un database, tenere traccia delle sessioni utenti, gestire i cookie.

Possibili ambiti di utilizzo dei servlet sono il supporto a richieste multiple in modo concorrente (es. conferenze on-line), l'interazione con database remoti attraverso il protocollo HTTP, le applicazioni di e-commerce, l'inoltro di richieste ad altri server per bilanciare il carico di lavoro tra diversi server che sono il mirror di un server principale o per suddividere i dati di un unico server logico tra più server fisicamente diversi. Possono essere visti come applet che girano su server. Permettono la distribuzione di contenuti dinamici in un modo portabile ed indipendente dalla piattaforma e dal Web server. Un'applicazione browser-based che chiama servlet non ha bisogno di supportare Java perché l'output di un servlet può essere HTML od XML o qualsiasi altro tipo di contenuto. In passato una grossa limitazione alla diffusione dei servlet era data dal fatto che l'unico Web server in grado di eseguirli era Java Web Server. In seguito sono stati sviluppati diversi servlet engine che hanno la possibilità di collegarsi ai Web server più comuni rendendoli compatibili con i servlet. Il più diffuso tra i servlet engine è JRun (Java Servlet Runner).

Si rimanda al Cap.6 per una descrizione dettagliata di tale tecnologia.

5.2.5.2 La tecnologia JavaServer Pages

La tecnologia JavaServer Pages (JSP) fu disegnata per fornire un metodo di sviluppo di servlet dichiarativo e presentation-centric. Oltre tutti i vantaggi offerti dai servlet, tale tecnologia offre la capacità di sviluppare rapidamente servlet dove il contenuto e la display logic sono separati, e di riusare il codice attraverso un'architettura component-based.

Sia servlet sia pagine JSP descrivono come processare una richiesta (da un client HTTP) per creare una risposta. Mentre i servlet sono espressi in Java, le pagine JSP sono **documenti testuali** (text-based) che includono una combinazione di HTML, tag JSP e codice Java. Sebbene entrambe possano essere usate per risolvere gli stessi problemi, ciascuna è destinata a compiere specifici processi. La tecnologia Servlet è stata sviluppata come meccanismo per accettare richieste da browser, reperire dati da database o altre fonti nel back-end, realizzare la logica applicativa sui dati (specialmente nei casi in cui il servlet accede direttamente al database), e formattare tali dati per la presentazione nel browser (di solito in HTML). Un servlet usa istruzioni di print per inviare codice HTML al browser. Incorporare HTML in istruzioni print causa due problemi. Primo, i Web designer non possono prevedere il look di una pagina HTML fino all'esecuzione. Secondo, quando i dati od il loro formato di visualizzazione cambiano, localizzare le sessioni appropriate di codice nel servlet è molto difficile. In aggiunta, quando la presentation logic ed il contenuto sono mischiati, i cambiamenti nel contenuto richiedono che un servlet sia ricompilato e ricaricato sul Web server. Le pagine JSP forniscono un meccanismo per specificare il mapping da un componente JavaBeans al formato di presentazione HTML (o XML). Dato che le pagine JSP sono text-based, un Web designer può usare strumenti di sviluppo grafici per creare e visualizzare il loro contenuto. Gli stessi tool possono essere usati per specificare dove sono visualizzati i dati provenienti dall'EJB Tier o dall'EIS Tier. Le pagine JSP usano Java come linguaggio di scripting.

Quando un Web designer modifica una pagina JSP, tale pagina è automaticamente ricompilata e ricaricata sul Web server. Inoltre, tutte le pagine JSP di un'applicazione Web possono essere compilate prima del deployment per una migliore efficienza.

JSP permette quindi di definire chiaramente cosa è application logic e cosa è contenuto. Come i servlet, la tecnologia JSP è un modo efficiente di fornire contenuti dinamici in maniera portabile e platform o application-independent. Inoltre supporta un modello a componenti riusabili attraverso l'inclusione della tecnologia JavaBeans e le tag library.

Riassumendo, JSP fornisce un modo facile per sviluppare contenuti dinamici servlet-based, con il vantaggio aggiuntivo di separare il contenuto dalla display logic, permettendo la suddivisione dei ruoli per specifiche competenze.

Si rimanda al Cap.7 per una descrizione dettagliata di tale tecnologia.

5.3 Ruoli dei Web Component

Nel modello di programmazione delle applicazioni J2EE i componenti Web possono svolgere due ruoli: **Presentation Component** o **Front Component**. La Figura 5.2 illustra il meccanismo di base. Il FC accetta una richiesta, poi determina il PC appropriato cui inoltrarla. Il PC processa la richiesta e ritorna la risposta al FC, che la rispedisce al server per la presentazione all'utente.

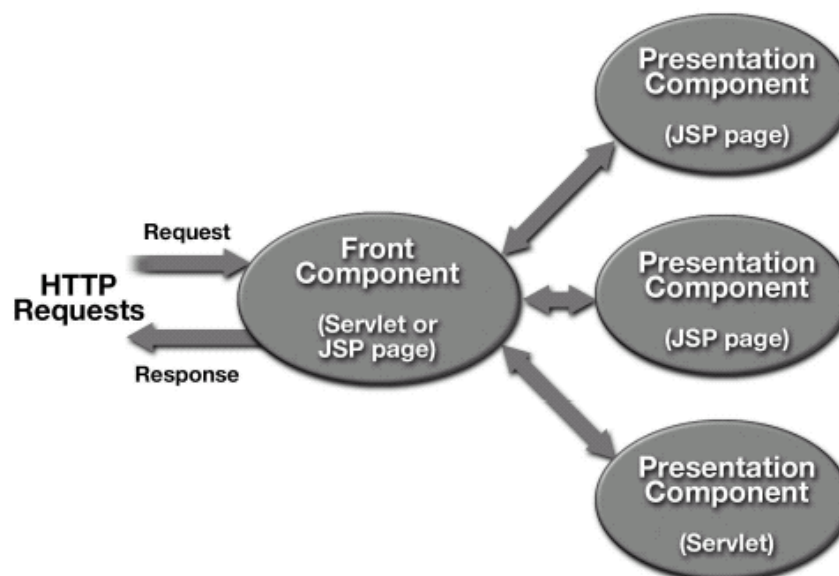


Figura 5.2 Ruoli dei Web Component

5.4 Application Design

Esistono molti modi per disegnare un'applicazione Web. Con la piattaforma J2EE possono essere implementati quattro tipi generali di applicazioni Web: "basic HTML", "HTML with basic JSP pages and servlets", "JSP pages with JavaBeans components", e "highly-structured applications with modular components e enterprise beans". I primi tre tipi di applicazioni sono considerati Web-centric, il quarto EJB-centric. La scelta del tipo di applicazione dipende ovviamente da vari fattori, tra i quali particolare importanza rivestono vincoli di tempo, complessità del problema, capacità e risorse del team di sviluppo, longevità di un'applicazione, dinamismo dei contenuti da gestire e proporre in un'applicazione. Nella Figura 5.3 viene presentata in modo schematico tutta la gamma di applicazioni Web ed il loro uso in relazione a due fattori principali: complessità e robustezza.

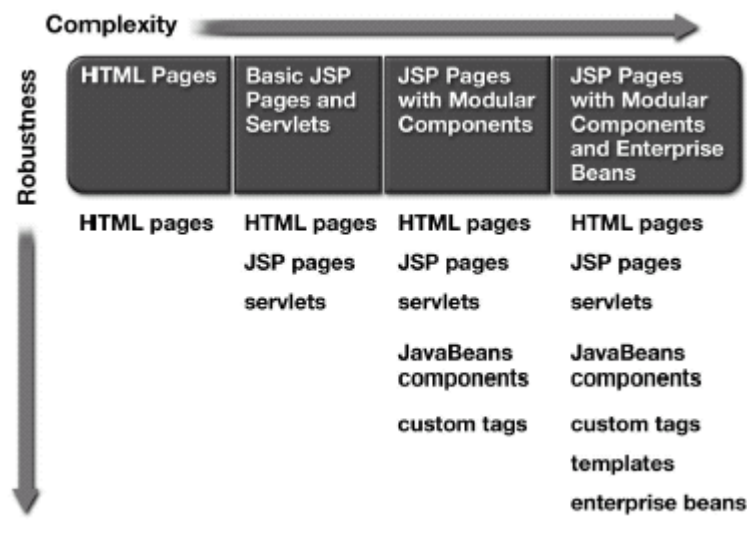


Figura 5.3 Application Design

5.4.1 Applicazioni con Pagine JSP base e Servlet

Tali applicazioni sono simili, come complessità, alle convenzionali applicazioni basate su HTML e CGI largamente deployate sul Web, tranne che le parti dinamiche delle pagine e l'interazione con l'utente sono gestite da JSP o servlet al posto degli script CGI. Applicazioni HTML con pagine JSP base sono applicazioni Web entry-level con molta della loro logica in servlet o pagine JSP. Possono essere sviluppate rapidamente, ma sono molto difficili da estendere e mantenere. In queste semplici applicazioni, alcune pagine visualizzano contenuti HTML statici. Dove è necessario mostrare contenuti dinamici (es. contenuto generato usando dati da un database), una pagina JSP o un servlet dovrebbero contenere il codice per connettersi al database e recuperare i dati.

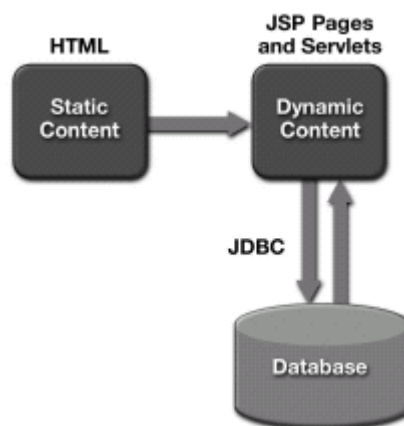


Figura 5.4 Applicazione con pagine JSP base e Servlet

Il layout di tali applicazioni non cambierà frequentemente. Il contenuto usato per il layout di pagina sarà legato all'applicazione. I cambiamenti alle pagine dinamiche, quindi, potranno essere apportati solo da esperti familiari con il linguaggio di programmazione Java. Includere gran parte della logica nelle pagine JSP o in servlet va bene per prototipizzare un'applicazione o per ambienti controllati quali siti Intranet, dove l'applicazione non è generalmente usata da un largo numero di utenti. Al crescere della complessità, si deve far ricorso ad un modello che permette una maggior modularizzazione dei componenti.

5.4.2 Applicazioni con Componenti Modulari

Quando si sviluppano applicazioni con contenuti dinamici ed un alto grado di interattività con l'utente, si dovrebbero usare pagine JSP con componenti JavaBeans e tag library. Questi tipi di componenti possono essere usati per generare contenuti, processare richieste, e visualizzare contenuti personalizzati. La Figura 5.5 mostra il cammino che un utente dovrebbe percorrere attraverso un'ipotetica applicazione Web interattiva, e mostra come componenti riutilizzabili possano essere usati ad ogni passo nel processo.



Figura 5.5 Pagina JSP con componenti modulari

Processare le richieste di un utente è un aspetto molto importante del comportamento di un'applicazione Web, e può essere implementato efficientemente usando componenti modulari. Le applicazioni saranno così più facili da sviluppare e mantenere. La Figura 5.6 mostra come i dati provenienti da un form possono essere processati in un'applicazione Web. In questo esempio, relativo alla creazione di un account utente, un utente invia i dati sul suo account da un browser. Tali dati sono inoltrati al Process Request Bean, che li elabora convertendoli in dati di account mantenuti da un componente JavaBeans Account Bean. Tali dati sono memorizzati in un database usando una connessione JDBC ottenuta dal JDBC Connection Pool Bean. Se i dati sono stati inseriti correttamente, il Process Request Bean invia all'utente la pagina appropriata confermando la creazione dell'account.

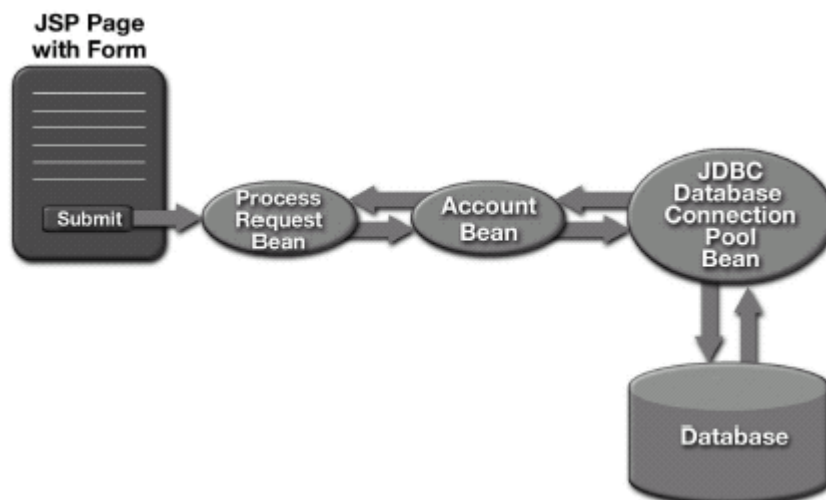


Figura 5.6 Processare una richiesta con componenti riutilizzabili

Per evitare confusione, i componenti JavaBeans che interagiscono con l'utente ed i dati esterni (in questo esempio, il bean che processa la richiesta) dovrebbero essere separati dai componenti che rappresentano i dati (l'Account Bean). Questa separazione tra contenuti e dati permette ai componenti di essere riutilizzati e all'applicazione di essere migrata verso un disegno più complesso se dovesse cambiare il suo ambito. Questa tipologia di applicazione vede una chiara separazione della business logic dalla display logic. Il contenuto sarà più facile da modificare ed i componenti, se ben progettati, saranno, come detto, riutilizzabili. La maggior debolezza di questa soluzione è dovuta alla necessità per gli sviluppatori di fornire connessioni ad applicazioni legacy e supporto alle transazioni. Appena aumenta la complessità dell'applicazione e diventano fondamentali il bisogno di maggior supporto transazionale e di integrazione con le risorse esterne, si richiede un approccio più strutturato.

5.4.3 Applicazioni EJB-Centric

Un'applicazione EJB-Centric estende le applicazioni modulari basate su componenti, con due differenze. Primo, questo design usa un Front Component come Controller. Secondo, i dati rappresentati dai componenti JavaBeans sono mantenuti dagli enterprise bean. Questo design fornisce flessibilità, manageability e separazione delle responsabilità tra gli sviluppatori. La flessibilità è fornita usando un'architettura MVC unita ad un Front Component. L'architettura MVC permette una chiara separazione della business logic, dei dati e della presentation logic. La Figura 5.7 mostra come un'architettura MVC può essere implementata usando pagine JSP, servlet, componenti JavaBeans e componenti EJB.

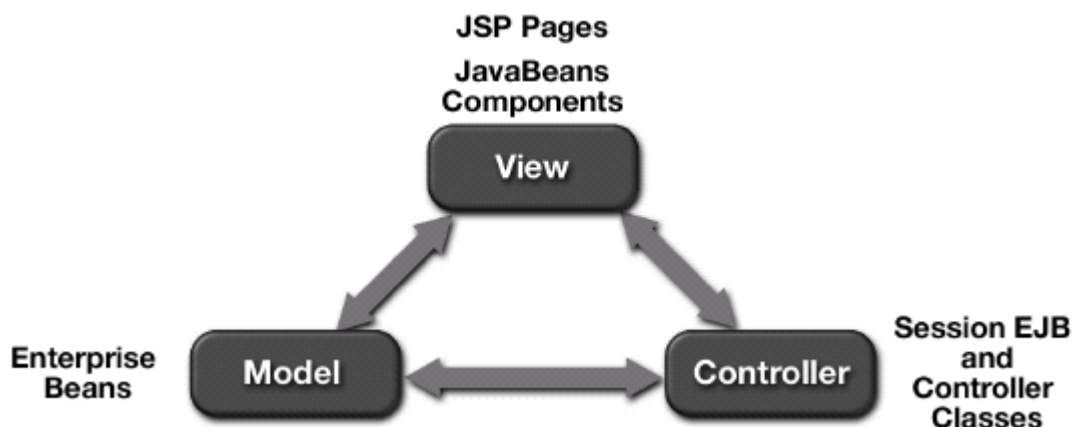


Figura 5.7 Architettura MVC per un'applicazione EJB-centric

Come illustrato nella figura, la logica che guida l'applicazione è separata dalla presentation logic e dai dati presentati all'utente. In questo design un Controller centrale riceve tutte le richieste ed aggiorna i componenti JavaBeans che contengono la View dei dati.

5.5 Riassunto

Il Web richiede agli sviluppatori di creare interfacce che siano flessibili e facili da mantenere. Le applicazioni Web possono essere rese più flessibili e facili da mantenere tramite le tecnologie a componenti della piattaforma J2EE quali servlet e JSP, usate per generare contenuti dinamici in maniera portabile e scalabile. Tali tecnologie sono la risposta di Sun ai problemi intrinseci di altre tecnologie progettate per la creazione di contenuti dinamici, quali CGI, ISAPI, NSAPI e ASP.

Dipendentemente da vincoli di tempo, complessità del problema, capacità e risorse del team di sviluppo, longevità di un'applicazione, dinamismo dei contenuti da gestire e proporre, cambierà la scelta del tipo di applicazione. Le applicazioni Web enterprise dovrebbero essere sviluppate usando componenti modulari, che includono servlet, pagine JSP, componenti JavaBeans e tag library. Le architetture per applicazioni Web includono pagine JSP base e servlet, applicazioni Web-centric che usano pagine JSP con componenti modulari, ed applicazioni EJB-centric che usano pagine JSP con enterprise bean.

Capitolo 6 La tecnologia Servlet

6.1 Overview

6.1.1 Che cos'è un Servlet?

Un servlet è un componente Web basato su tecnologia Java, indipendente dalla piattaforma, compilato in bytecode platform neutral e caricato ed eseguito da un Web server che lo utilizzerà come propria estensione. Il Web server di fatto mette a disposizione dei servlet un container (o servlet engine), un'estensione del Web server, che si occuperà della gestione del loro ciclo di vita, della gestione dell'ambiente all'interno del quale i servlet girano, dei servizi di sicurezza. I servlet interagiscono con i client attraverso un paradigma request/response implementato dal servlet container, che è quindi responsabile del passaggio dei dati dal client verso i servlet e, viceversa, del ritorno al client dei dati prodotti dall'esecuzione dei servlet.

Dal momento che un servlet è un oggetto server-side, può accedere a tutte le risorse messe a disposizione dal server per generare pagine dai contenuti dinamici come prodotto dell'esecuzione della business logic. La tecnologia servlet è supportata da una moltitudine di server Web: Apache, Netscape FastTrack, Netscape Enterprise, Microsoft IIS, IBM Internet Connection Server, Lotus Domino Go Webserver.

6.1.2 Che cosa è un Servlet Container?

Il servlet container è una parte di un Web server, o di un Application Server, che fornisce i servizi di rete sulla quale le richieste e le risposte sono inviate, decodifica richieste basate su contenuti MIME, e formatta risposte basate su MIME. Un servlet container inoltre contiene e gestisce i servlet durante il loro ciclo di vita.

Tutti i servlet container devono supportare HTTP come protocollo request/response, e opzionalmente protocolli come HTTPS (HTTP over SSL). La versione minima delle specifiche HTTP che un container deve implementare è HTTP/1.0. J2SE 1.2 è, invece, la versione minima della piattaforma Java sottostante con la quale i servlet container devono essere costruiti.

6.1.2.1 Nota sui Tipi MIME

MIME (Multipurpose Internet Mail Extensions) fu sviluppato come un sistema di specifica e descrizione del contenuto di un messaggio di posta elettronica. Con il tempo questo modello si è esteso ad altri usi, in particolare al Web, divenendo un sistema di specifica e descrizione del contenuto di un oggetto inviato da un server HTTP al browser. Le caratteristiche di MIME sono state definite inizialmente tramite l'RFC 1341. In seguito sono state aggiornate tramite le RFC 1521, 1522 e 1523. Attualmente sono di riferimento l'RFC 1896, e le RFC dalla 2045 alla 2049. Il tipo MIME di un documento è indicato nel campo **Content-Type**, in modo che il destinatario possa individuare il tipo di file trasmesso. I tipi MIME sono nella forma tipo/sottotipo. Alcuni esempi:

- tipo MIME multipart (utilizzato nei messaggi di posta elettronica contenenti allegati).
- tipo MIME application: application/msword, application/pdf, application/zip.
- tipo MIME audio: audio/x-midi, audio/x-wav, audio/mpeg.
- tipo MIME image: image/gif, image/jpeg, image/png, image/tiff.
- tipo MIME text: text/html; text/plain (txt), text/sgml, text/richtext (rtf), text/xml.
- tipo MIME video: video/mpeg, video/quicktime, video/x-msvideo (per il formato avi).

6.1.3 Una sequenza di esempio

Quella che segue è una tipica sequenza di eventi in cui è coinvolto un servlet:

1. Un client (es. un browser Web) accede ad un Web server e fa una richiesta HTTP.

2. La richiesta è ricevuta dal server Web e consegnata al servlet container. Questo può girare nello stesso processo del server Web, in un processo differente sullo stesso host, o su un host diverso da quello del server Web per il quale il container processa la richiesta.
3. Il container determina quale servlet invocare basandosi sulla configurazione dei suoi servlet, e lo chiama passandogli degli oggetti rappresentanti la richiesta e la risposta.
4. Il servlet usa l'oggetto richiesta per calcolare chi è l'utente remoto, quali parametri HTTP sono stati inviati come parte della richiesta, ed altri dati rilevanti. Il servlet realizza qualunque logica per la quale è stato programmato, e genera dati da rispedire al client attraverso l'oggetto risposta.
5. Una volta che il servlet ha finito di processare la richiesta, il servlet container assicura che la risposta sia inviata correttamente, e ritorna il controllo al Web server.

6.1.4 Confronto tra servlet ed altre tecnologie

Per quanto riguarda le funzionalità, i servlet giacciono tra programmi CGI ed estensioni server proprietarie quali Netscape Server API (NSAPI) o Apache Modules. Rispetto a CGI e agli altri meccanismi di estensione dei server la tecnologia servlet offre diversi vantaggi.

Una differenza sostanziale tra servlet e CGI è il meccanismo di esecuzione: i Servlet non richiedono la creazione di un nuovo processo, consentendo un utilizzo più efficiente delle risorse messe a disposizione dal server e risultando più veloci degli script CGI. Un servlet prevede un unico caricamento al momento della prima esecuzione; una volta attivato ed inizializzato, il servlet rimane in memoria e può essere chiamato innumerevoli volte per soddisfare le richieste provenienti dai client, scalando bene senza richiedere hardware addizionale. L'esecuzione del servlet termina solo quando viene invocato un particolare metodo che provvede a rilasciare la memoria occupata e concludere l'esecuzione. Una volta caricato in memoria, un servlet può girare su un thread leggero mentre gli script CGI devono essere caricati su un processo diverso per ogni richiesta.

Altro vantaggio è che, al contrario di uno script CGI, un servlet permette il pool di connessioni a database o ad altri oggetti Java: ciò comporta un evidente risparmio di tempo nell'elaborazione della richiesta.

Essendo scritti in Java, possono sfruttare tutti i vantaggi offerti da tale linguaggio: facilità di sviluppo, sicurezza, robustezza e portabilità. I servlet sono supportati da qualsiasi piattaforma che abbia una JVM ed un Web server che supporta servlet. Possono essere usati su differenti piattaforme senza essere ricompilati. Possono usare APIs generiche come JDBC per comunicare direttamente con risorse esistenti. Questo semplifica e velocizza lo sviluppo delle applicazioni Web. Essendo basati su Java, i servlet sono poi estendibili. Questo permette agli sviluppatori di estendere le funzionalità di una applicazione Web proprio come farebbero per una applicazione Java.

La tecnologia Servlet inoltre elimina gran parte della complessità del codice necessario per ottenere ed elaborare i parametri da una richiesta HTTP, complessità intrinseca delle applicazioni CGI-based, accedendo direttamente ai parametri tramite opportuni metodi.

Infine, uno dei vantaggi più grandi è che i servlet forniscono APIs per il session tracking di un'applicazione Web e per interagire con le richieste dell'utente. Il concetto di sessione è molto importante e permette di superare i limiti delle applicazioni Web dovuti alla natura stateless del protocollo HTTP.

6.2 L'interfaccia Servlet

L'interfaccia **Servlet** è l'astrazione centrale della API Servlet.

Tutti i servlet implementano questa interfaccia sia direttamente, oppure, più comunemente, estendendo una classe che la implementa. Le due classi della API Servlet che implementano tale interfaccia sono **GenericServlet** (deriva direttamente dalla classe **Object**) e **HttpServlet** (realizza servlet che sono in grado di utilizzare il protocollo HTTP per ricevere e spedire informazioni ad un browser).

6.2.1 Il package javax.servlet

Questo package è il package di base della API Servlet, e contiene le classi per definire servlet standard indipendenti dal protocollo. Tecnicamente un servlet generico è una classe definita a partire dall'interfaccia **Servlet** contenuta all'interno del package **javax.servlet**. Questa interfaccia contiene i prototipi di tutti i metodi necessari alla esecuzione delle logiche di business, nonché alla gestione del ciclo di vita dell'oggetto dal momento del suo istanziamento, sino al momento della sua terminazione.

```
package javax.servlet;
import java.io.*;
public interface Servlet {
    public abstract void destroy( );
    public ServletConfig getServletConfig( );
    public String getServletInfo( );
    public void service (ServletRequest req, ServletResponse res) throws IOException, ServletException;
}
```

Esempio 6.1 Prototipo dell'interfaccia Servlet

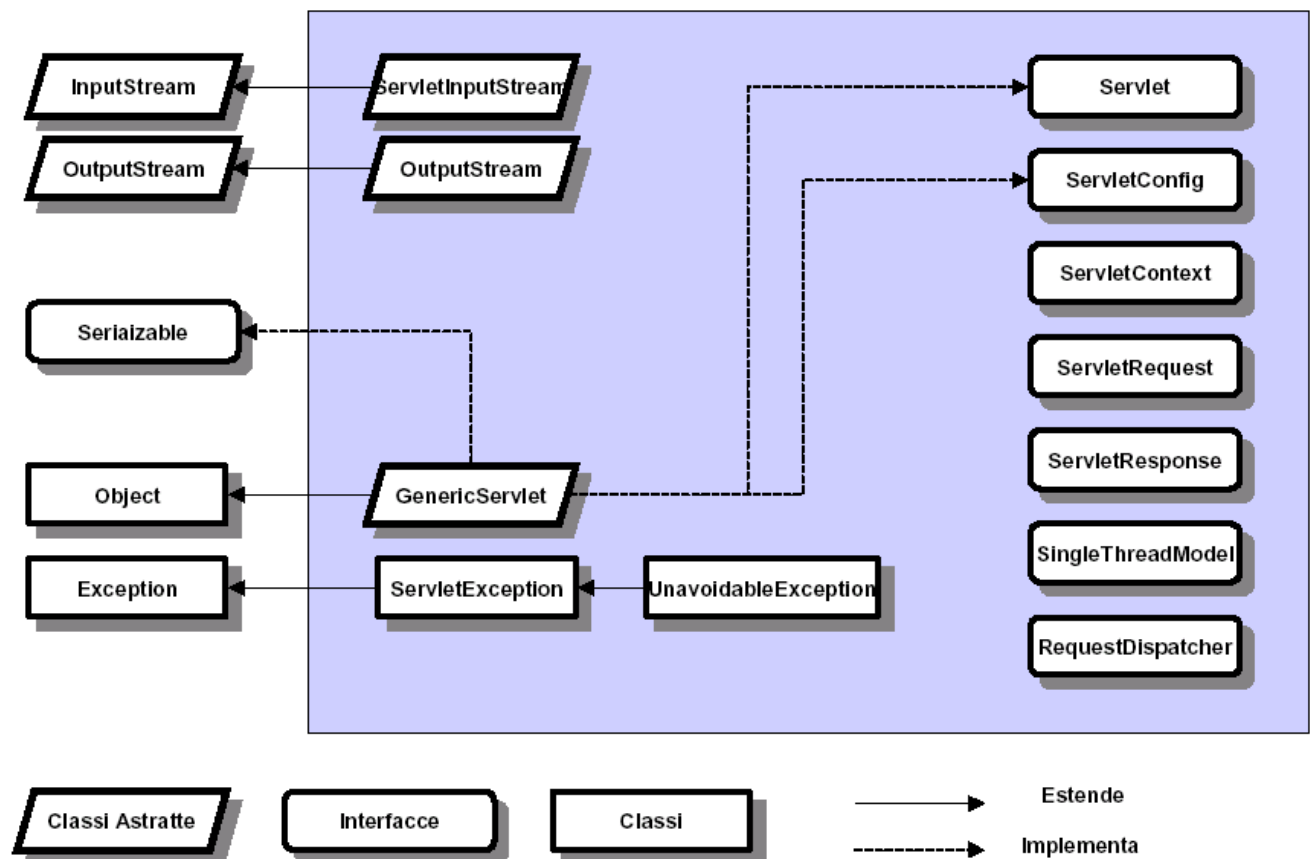


Figura 6.1 Il package javax.servlet

I metodi definiti nell'interfaccia Servlet devono essere supportati da tutti i servlet o possono essere ereditati attraverso la classe astratta **GenericServlet**, che rappresenta un'implementazione base di un servlet generico. Nella Figura 6.1 viene schematizzata la gerarchia di classi di questo package. Esso include inoltre una serie di classi utili alla comunicazione tra client e server, nonché alcune interfacce che definiscono i prototipi di oggetti utilizzati per tipizzare le classi che saranno necessarie alla specializzazione della servlet generica in servlet dipendenti da un particolare protocollo.

6.2.2 Il package javax.servlet.http

Il package **javax.servlet.http** supporta lo sviluppo di servlet che, specializzando la classe base astratta **GenericServlet** definita nel package **javax.servlet**, utilizzano il protocollo HTTP. Le classi di questo package estendono le funzionalità di base di un servlet supportando tutte le caratteristiche della trasmissione di dati con protocollo HTTP compresi cookie, richieste e risposte HTTP nonché metodi HTTP (GET, POST, HEAD, PUT, ecc...).

Nella Figura 6.2 è schematizzata la gerarchia del package in questione. Formalmente quindi, un servlet specializzato per generare contenuti specifici per il Web sarà ottenibile estendendo la classe base astratta **javax.servlet.http.HttpServlet**.

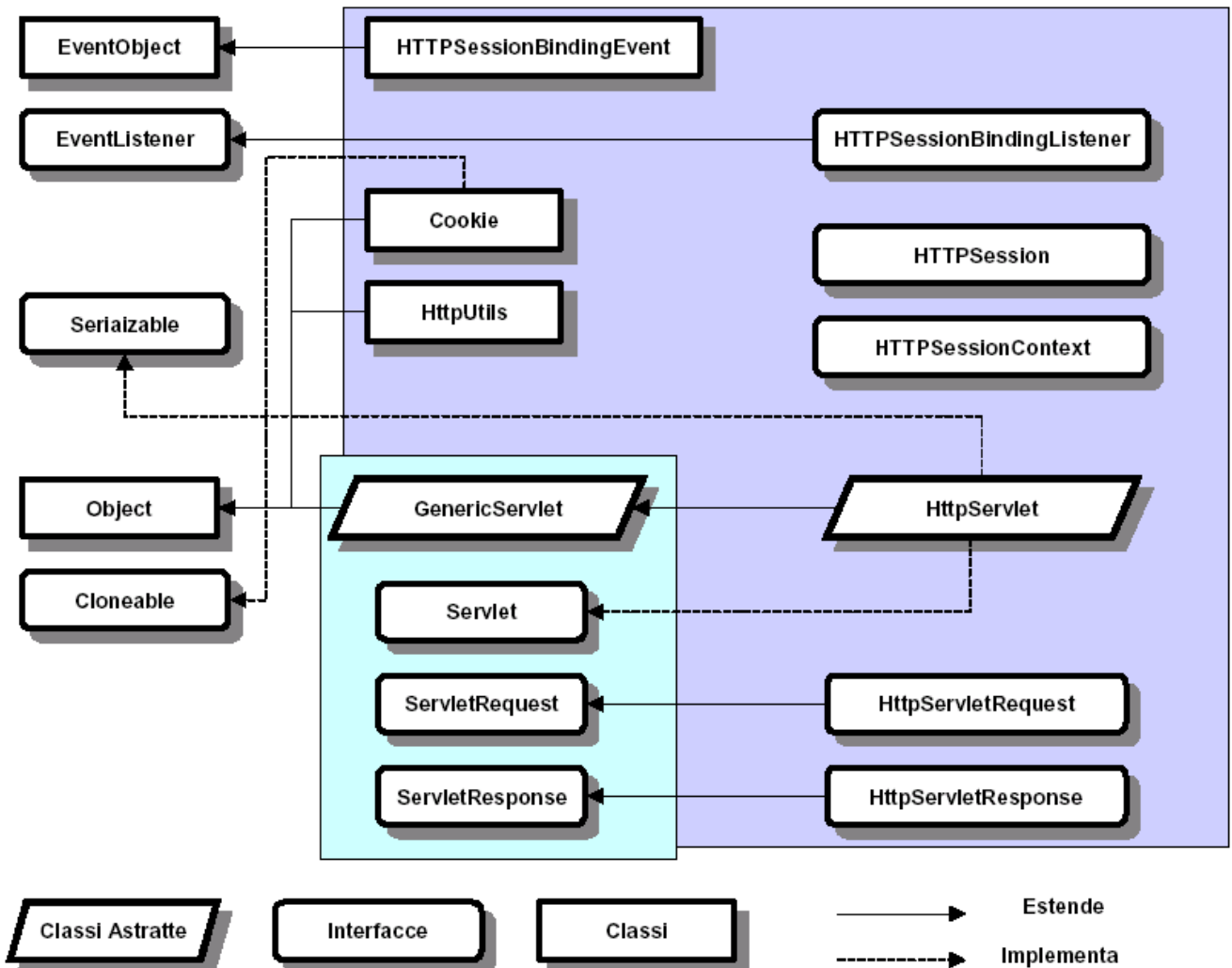


Figura 6.2 Il package javax.servlet.http

6.2.3 Metodi per la gestione delle richieste

L'interfaccia base **Servlet** definisce il metodo **service** per gestire le richieste dei client. Questo metodo è chiamato ad ogni richiesta che il servlet container inoltra ad un'istanza di un servlet.

La gestione di richieste concorrenti ad un'applicazione Web richiede allo sviluppatore di disegnare i servlet con thread multipli in esecuzione contemporaneamente all'interno del metodo **service**. Generalmente, il Web container gestisce richieste concorrenti allo stesso servlet attraverso l'esecuzione concorrente del metodo **service** su differenti thread.

Nella sezione successiva sono introdotti i metodi usati per gestire richieste specifiche basate sul protocollo HTTP.

6.2.3.1 Metodi per gestire specifiche richieste HTTP

La sottoclasse astratta **HttpServlet** aggiunge metodi aggiuntivi all'interfaccia base Servlet, i quali sono automaticamente chiamati dal metodo `service` e sono specifici per richieste HTTP:

- ◆ **doGet** per gestire richieste HTTP GET
- ◆ **doPost** per gestire richieste HTTP POST
- ◆ **doPut** per gestire richieste HTTP PUT
- ◆ **doDelete** per gestire richieste HTTP DELETE
- ◆ **doHead** per gestire richieste HTTP HEAD
- ◆ **doOptions** per gestire richieste HTTP OPTIONS
- ◆ **doTrace** per gestire richieste HTTP TRACE

Inoltre l'interfaccia `HttpServlet` definisce il metodo **getLastModified** per supportare operazioni GET condizionali, le quali richiedono l'invio di una risorsa solo se questa è stata modificata a partire da un certo istante. In particolari situazioni, l'implementazione di questo metodo può permettere un migliore e più efficiente utilizzo delle risorse di rete (Web caching).

6.2.4 Numero di istanze: Servlet e multithreading

La dichiarazione del servlet, che è una parte del deployment descriptor dell'applicazione Web contenente il servlet, controlla come il servlet container fornisce le istanze del servlet. Per un servlet non residente in un ambiente distribuito, il servlet container usa solo un'istanza per ogni dichiarazione di servlet. Per un servlet che implementa l'interfaccia `SingleThreadModel`, il servlet container può istanziare istanze multiple per gestire un carico di richieste pesante e serializzare le richieste ad una particolare istanza.

La situazione tipica prevede comunque che quando n richieste da parte dei client arrivano al Web server, vengono creati n thread differenti in grado di accedere ad un particolare servlet in maniera concorrente (Figura 6.3).

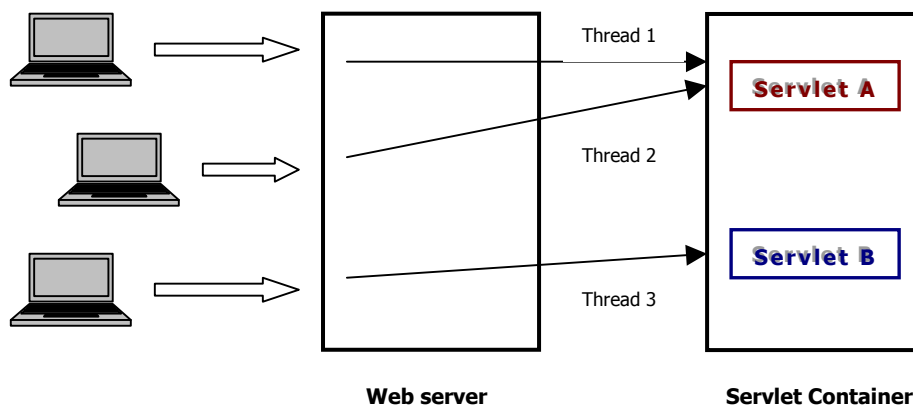


Figura 6.3 Accessi concorrenti a servlet: una istanza per ogni servlet

Come tutti gli oggetti Java, i servlet **non sono oggetti thread-safe**, in altre parole è necessario che il programmatore definisca le politiche di accesso all'istanza della classe da parte del thread. Inoltre, lo standard definito da Sun prevede che un server possa utilizzare una sola istanza di questi oggetti (questa limitazione anche se tale aiuta all'ottimizzazione della gestione delle risorse per cui, ad esempio, una connessione ad un database sarà condivisa tra tante richieste da parte di client). Mediante l'utilizzo dell'operatore **synchronized**, è possibile sincronizzare l'accesso alla classe da parte dei thread dichiarando il metodo `service` di tipo sincronizzato o limitando l'utilizzo del modificatore a singoli blocchi di codice che contengono dati sensibili.

6.2.4.1 L'interfaccia `SingleThreadModel`

Quando un thread accede al metodo sincronizzato di un servlet, ottiene il lock sull'istanza dell'oggetto. Abbiamo detto che un Web server per definizione utilizza una sola istanza di servlet condividendola tra le varie richieste da parte dei client. Stiamo creando un collo di bottiglia, che

in caso di sistemi con grossi carichi di richieste potrebbe ridurre significativamente le prestazioni del sistema. Se il sistema dispone di abbondanti risorse, la API Servlet ci mette a disposizione un metodo per risolvere il problema, a scapito delle risorse della macchina, rendendo le classi servlet **thread-safe**. Formalmente, un servlet viene considerato thread-safe se implementa l'interfaccia **javax.servlet.SingleThreadModel**. In questo modo saremo sicuri che solamente un thread avrà accesso ad una istanza della classe in un determinato istante, cioè l'uso di tale interfaccia garantisce che solo un thread alla volta eseguirà il metodo service di una istanza di un dato servlet. A differenza dell'utilizzo del modificatore synchronized, in questo caso il Web server creerà più istanze di uno stesso servlet (tipicamente in numero limitato e definito) al momento del caricamento dell'oggetto, e utilizzerà le varie istanze assegnando al thread che ne faccia richiesta la prima istanza libera (Figura 6.4), e riuscendo così a gestire un carico di richieste maggiore e a serializzare le richieste ad una particolare istanza.

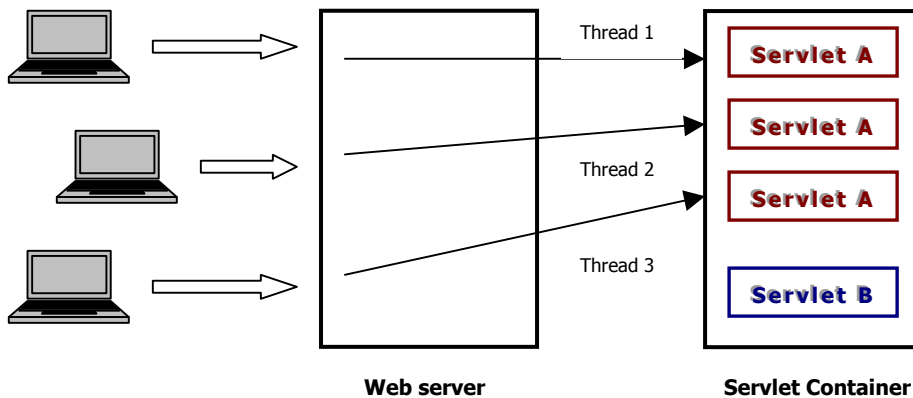


Figura 6.4 SingleThreadModel: più istanze per ogni servlet

6.2.5 Il ciclo di vita di un Servlet

Un servlet è gestito attraverso un ciclo di vita ben definito, il quale descrive come il servlet è caricato, istanziato e inizializzato, come gestisce richieste dai client, e come è rimosso dal servizio prima di passare sotto la responsabilità del garbage collector. Il life cycle è espresso dai metodi `init()`, `service()` e `destroy()` dell'interfaccia `javax.servlet.Servlet` che tutti servlet, come detto, devono implementare direttamente, o indirettamente attraverso le due classi astratte `GenericServlet` o `HttpServlet`.

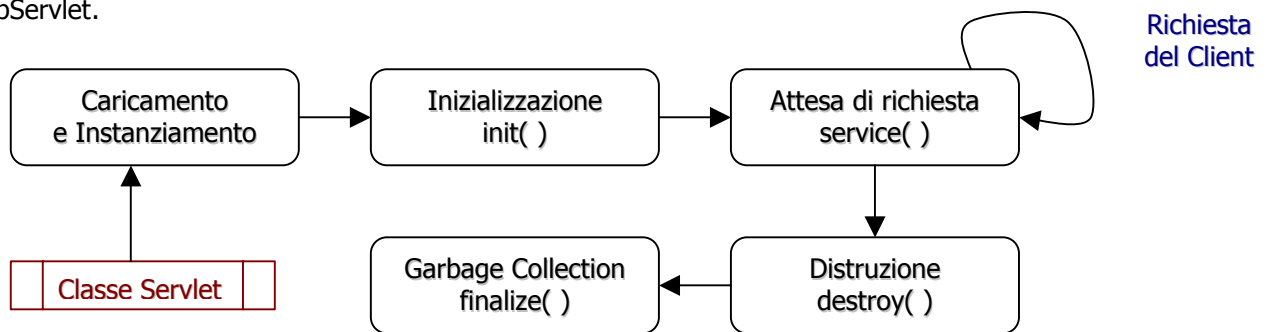


Figura 6.5 Ciclo di vita di un servlet

6.2.5.1 Caricamento ed Instanziamento

Il servlet container è responsabile del caricamento ed istanziamento dei servlet. Un servlet viene caricato ed istanziato o quando il container è avviato, o nel momento in cui il container determina che il servlet è necessario per servire una richiesta. Questa fase viene eseguita dal Web server utilizzando l'oggetto **Class** di `java.lang`. Il caricamento può avvenire da un file system locale, un file system remoto o un altro servizio di rete. Dopo aver caricato la classe servlet, il container la istanzia per l'uso.

6.2.5.2 Inizializzazione

Dopo che il servlet è stato istanziato, il container deve inicializzarlo prima che questo possa gestire le richieste dai client. In genere durante questa fase il servlet carica dati persistenti, apre connessioni (basate sull'API JDBC) verso database o stabilisce legami con altre entità esterne. L'inizializzazione del servlet avviene mediante la chiamata al metodo **init**, definito nell'interfaccia Servlet ed ereditato dalla classe base astratta **javax.servlet.http.HttpServlet**. Nel caso in cui il metodo non venga riscritto, il metodo ereditato non eseguirà alcuna operazione. Il metodo **init** di un servlet prende come parametro di input un oggetto di tipo **ServletConfig**, il quale consente di accedere a parametri di inicializzazione passati attraverso il Web server nella forma di coppie chiave-valore. Il metodo **init** viene eseguito un'unica volta, può non essere thread-safe e durante la sua esecuzione non sono gestite richieste.

Durante l'inizializzazione, l'istanza del servlet può generare una **UnavailableException** o una **ServletException**. In questo caso il servlet non deve essere avviato al servizio; piuttosto deve essere rilasciato dal servlet container. Il metodo **destroy** non è chiamato dato che si considera inicializzazione senza successo.

6.2.5.3 Gestione delle richieste

Dopo che un servlet è stato opportunamente inicializzato, il servlet container può usarlo per gestire richieste da un client. Queste sono rappresentate da oggetti di tipo **ServletRequest**. Il servlet si occupa delle risposte alle richieste chiamando i metodi dell'oggetto **ServletResponse**. Questi due oggetti sono passati come parametri al metodo **service** dell'interfaccia Servlet. Nel caso di una richiesta HTTP, e quindi di una risposta HTTP, gli oggetti forniti dal container sono di tipo **HttpServletRequest** e **HttpServletResponse**.

Il servlet gestisce richieste tramite il metodo **service**. Questo metodo gestisce la richiesta e produce la risposta, e deve essere thread-safe; in genere sarà chiamato più volte in concorrenza. Per non gestire la concorrenza tra thread, il servlet, come già accennato, deve implementare l'interfaccia **SingleThreadModel**.

Un servlet container può spedire richieste concorrenti attraverso il metodo **service** del servlet. L'elaborazione concorrente con thread multipli nel metodo **service** per gestire tali richieste è a carico dello sviluppatore.

Un'alternativa è implementare l'interfaccia **SingleThreadModel**, la quale richiede al container di garantire che ci sia un solo thread alla volta nel metodo **service**. Un container può soddisfare questo requisito o serializzando le richieste o mantenendo un pool di istanze servlet (per ogni JVM se il servlet è parte di un'applicazione Web distribuita).

Per servlet che non implementano l'interfaccia **SingleThreadModel**, se il servlet container non può usare l'approccio basato su pool di istanze deve serializzare le richieste sincronizzando il metodo **service** (o metodi come **doGet** o **doPost** che sono inviati al metodo **service** della classe astratta **HttpServlet**) tramite la keyword **synchronized**.

Un servlet può generare sia una **ServletException** sia una **UnavailableException** durante il servizio di una richiesta. Una **ServletException** segnala che è occorso qualche errore durante l'elaborazione di una richiesta e che il container dovrebbe prendere gli opportuni provvedimenti per ripulire la richiesta. Una **UnavailableException** segnala che il servlet non può gestire la richiesta, o temporaneamente o permanentemente (in quest'ultimo caso il container deve rimuovere il servlet dal servizio chiamando il suo metodo **destroy**, e rilasciare l'istanza del servlet).

6.2.5.4 Fine del servizio

Il servlet container non deve necessariamente lasciare un servlet caricato in memoria a tempo indeterminato. Un'istanza di un servlet potrebbe essere mantenuta attiva sia per un periodo di millisecondi, sia per l'intero lifetime del servlet container (che potrebbe essere giorni, mesi o anni), sia per qualsiasi intervallo di tempo tra questi estremi.

I servlet accettano richieste finché non sono rimossi dal servizio. Quando il servlet container determina che un servlet deve essere rimosso dal servizio (es. fase di shut-down del container), chiama il metodo **destroy** dell'interfaccia Servlet per permettere al servlet di rilasciare le risorse che stava usando e salvare ciascuno stato persistente. L'overriding di questo metodo consentirà di rilasciare tutte le risorse utilizzate dal servlet (es. connessioni a database), garantendo che il

sistema non rimanga in uno stato inconsistente a causa di una gestione malsana da parte dell'applicazione Web. Il metodo `destroy` non può andare in concorrenza con altri `destroy`, mentre può andare in concorrenza con altre richieste al metodo `service`.

Prima della chiamata del metodo `destroy`, il container deve permettere a tutti i thread che stanno in quel momento girando sul metodo `service` del servlet, di completare l'esecuzione, o di superare un limite temporale definito per quel servlet.

Una volta distrutta l'istanza del servlet, il container non le deve più instradare richieste. Se dovesse abilitare il servlet nuovamente, farà ciò con una nuova istanza della classe del servlet. Terminato il metodo `destroy`, il container deve rilasciare l'istanza del servlet che diventa idonea per la garbage collection.

6.2.6 Un primo esempio di classe Servlet

Il servlet di esempio proposto di seguito fornisce la versione Web della classica applicazione Java HelloWorld. Una volta chiamato, restituirà una pagina HTML contenente semplicemente la stringa HelloWorld.

```
import javax.servlet.* ;
import javax.servlet.http.* ;
public class HelloWorldServlet extends HttpServlet {
    public void service (HttpServletRequest req, HttpServletResponse res) throws ServletException, IOException {
        res.setContentType("text/html");
        ServletOutputStream out = res.getOutputStream( );
        out.println("<html>");
        out.println("<head><title>Hello World</title></head>"); // imposta il titolo della pagina HTML
        out.println("<body>");
        out.println("<h1>Hello World</h1>");
        out.println("</body></html>");
    }
}
```

Esempio 6.2 HelloWorldServlet.java

6.2.7 Una nota sul metodo `service`

Se il metodo `service` di un servlet non viene modificato, la nostra classe eredita di default il metodo `service` definito all'interno della classe astratta **HttpServlet** (Figura 6.6). Essendo il metodo chiamato in causa al momento dell'arrivo di una richiesta da parte di un client, nella sua forma originale questo metodo ha funzioni di dispatcher tra altri metodi basati sul tipo di richiesta HTTP in arrivo dal client.

Come già detto, il protocollo HTTP ha una varietà di tipi differenti di richieste che possono essere avanzate da parte di un client. Comunemente quelle di uso più frequente sono le richieste di tipo GET e POST. Nel caso di richiesta di tipo GET o POST, il metodo `service` definito all'interno della classe astratta `HttpServlet` chiamerà rispettivamente i metodi **doGet** o **doPost**, i quali conterranno ognuno il codice per gestire la particolare richiesta. In questo caso quindi, non avendo applicato la tecnica di overriding sul metodo `service`, sarà necessario implementare uno di questi metodi all'interno della nostra nuova classe secondo il tipo di richieste che dovrà esaudire.

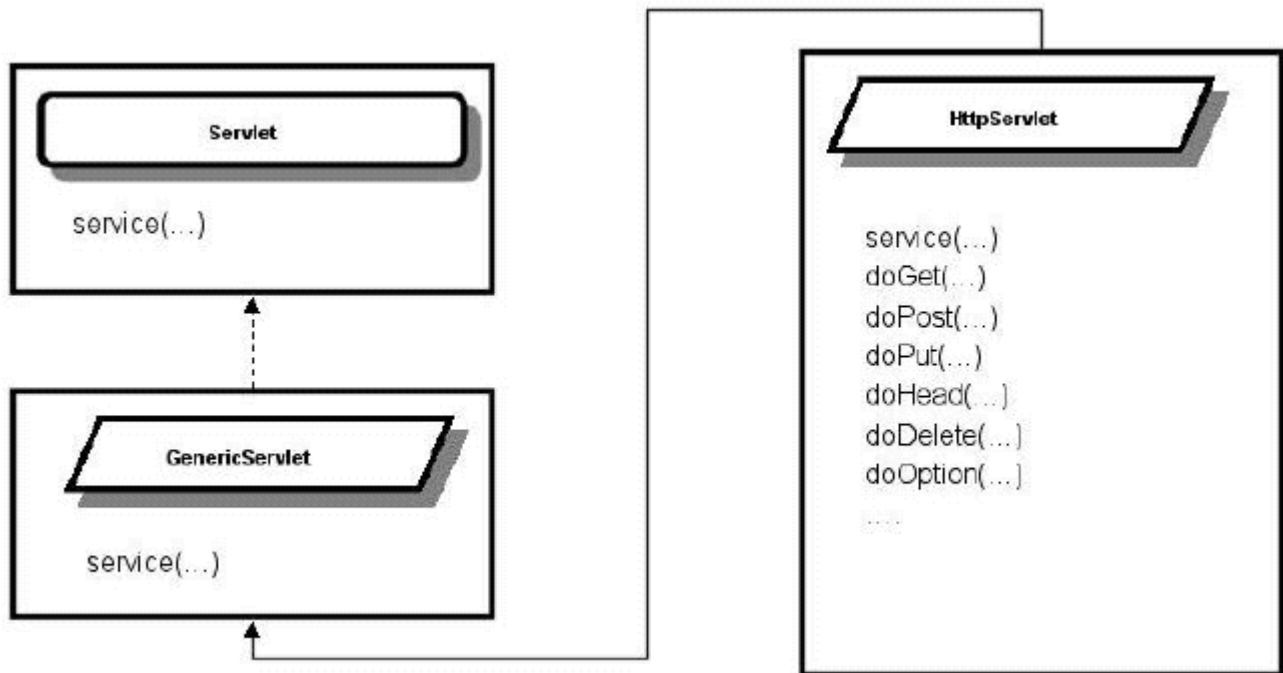


Figura 6.6 La classe astratta HttpServlet

6.3 L'interfaccia ServletContext

L'interfaccia **ServletContext** definisce una view di un servlet di un'applicazione Web entro la quale il servlet sta girando. Usando l'oggetto ServletContext un servlet può loggare eventi, ottenere riferimenti URL a risorse, e settare e memorizzare attributi ai quali potrebbero accedere altri servlet nel contesto. Esiste un'istanza dell'interfaccia ServletContext per ogni applicazione Web deployata in un container (ulteriormente per ogni JVM se il container è distribuito). I server Web possono supportare host logici multipli che condividono un unico indirizzo IP su un server. Questa capacità è nota come **virtual hosting**. In questo caso, ogni host logico deve avere il suo servlet context. I servlet context non possono essere condivisi su host virtuali.

I metodi **getInitParameter** e **getInitParameterNames** dell'interfaccia ServletContext permettono al servlet di accedere ai parametri di inizializzazione del contesto associati ad un'applicazione Web (es. indirizzo e-mail del Webmaster, nome di un sistema che memorizza dati critici) e locali alla JVM nella quale sono stati creati.

Un servlet può lavorare con un attributo entro il contesto per nome. Qualsiasi attributo contenuto entro un contesto è disponibile a tutti i servlet che sono parte della stessa applicazione Web. I metodi **setAttribute**, **getAttribute**, **getAttributeNames** e **removeAttribute** permettono di accedere ai vari attributi del contesto.

L'interfaccia fornisce inoltre accesso diretto alla gerarchia di documenti dal contenuto statico che sono parte dell'applicazione Web, inclusi file HTML ed immagini GIF e JPEG, attraverso i metodi **getResource**, **getResourceAsStream** e **getResourcePaths**. I primi due metodi prendono una String che inizia con un "/" ed indica il cammino della risorsa relativo alla radice del contesto. Questa gerarchia di documenti può esistere sul file system del server Web, sul file archivio (WAR) dell'applicazione Web, su un server remoto, o su altre location. E' importante notare che questi metodi non sono usati per ottenere contenuti dinamici. Così, in un container che supporta JSP, la chiamata `getResource("/index.jsp")` ritorna il codice sorgente della pagina JSP e non l'output opportunamente processato.

Per ogni contesto è richiesta una directory di memorizzazione temporanea; tale directory deve essere fornita dal container e resa disponibile attraverso un attributo di contesto particolare: **javax.servlet.context.tempdir**. Gli oggetti associati all'attributo devono essere di tipo **java.io.File**. Il container non deve mantenere il contenuto della directory temporanea quando viene riavviato, ma deve assicurare che il contenuto della directory temporanea di un contesto non sia visibile ai contesti di altre applicazioni Web che girano sul servlet container.

6.4 La Richiesta

L'oggetto richiesta incapsula tutte le informazioni della richiesta client. Nel protocollo HTTP, queste informazioni sono trasmesse dal client al server negli header HTTP e nel message body della richiesta.

```
package javax.servlet;
import java.net.*;
import java.io.*;
import java.util.*;
public interface ServletRequest {
    public Object getAttribute(String name);
    public Enumeration getAttributeNames( );
    public String getCharacterEncoding( );
    public int getContentLength( );
    public String getContentType( );
    public ServletInputStream getInputStream( ) throws IOException;
    public String getParameter(String name);
    public Enumeration getParameterNames( );
    public String[] getParameterValues(String name);
    public String getProtocol( );
    public BufferedReader getReader( ) throws IOException;
    public String getRealPath(String path);
    public String getRemoteAddr( );
    public String getRemoteHost( );
    public String getScheme( );
    public String getServerName( );
    public int getServerPort( );
    public Object setAttribute(String name, Object attribute);
}
```

Esempio 6.3 L'interfaccia ServletRequest

```
package javax.servlet.http;
import java.util.*;
import java.io.*;
public interface HttpServletRequest extends javax.servlet.ServletRequest {
    String getAuthType( );
    Cookie[] getCookies( );
    long getDateHeader(String name);
    String getHeader(String name);
    Enumeration getHeaderNames( );
    int getIntHeader(String name);
    String getMethod( );
    String getPathInfo( );
    String getPathTranslated( );
    String getQueryString( );
    String getRemoteUser( );
    String getRequesteSessionId( );
    String getRequestURI( );
    String getServletPath( );
    HttpSession getSession( );
    HttpSession getSession(boolean create);
    boolean isRequestedSessionIdFromCookie( );
    boolean isRequestedSessionIdFromUrl( );
    boolean isRequestedSessionIdFromURL( );
    boolean isRequestedSessionIdValid( );
}
```

Esempio 6.4 L'interfaccia HttpServletRequest

Come oggetti di tipo **HttpServletResponse** rappresentano una risposta HTTP, oggetti di tipo **HttpServletRequest** rappresentano una richiesta HTTP. Mediante i metodi messi a disposizione da questa interfaccia, è possibile accedere ai contenuti della richiesta HTTP inviata dal client, compresi eventuali parametri o entità trasportate all'interno del pacchetto HTTP. Per esempio, i metodi **getInputStream** e **getReader** permettono di accedere ai dati trasportati dal protocollo. Il metodo **getParameter** ci consente di ricavare i valori dei parametri contenuti all'interno della query string della richiesta referenziandoli tramite il loro nome.

6.4.1 I parametri del protocollo HTTP

I parametri di una richiesta per un servlet sono stringhe mandate dal client al servlet container come parte della sua richiesta. Sono memorizzati come coppie nome-valore. Possono esistere valori multipli per ciascun nome di parametro. I metodi **getParameter**, **getParameterNames** e **getParameterValues** (che ritorna un array di String contenenti tutti i valori associati ad un nome di un parametro) dell'interfaccia `ServletRequest` sono disponibili per accedere ai parametri di una richiesta.

6.4.2 Attributi

Gli attributi sono oggetti associati ad una richiesta. Possono essere impostati sia dal container per esprimere informazioni che non potrebbero essere altrimenti espresse via API, sia da un servlet per comunicare informazioni ad un altro servlet (tramite il **RequestDispatcher**). Gli attributi sono acceduti attraverso i metodi **getAttribute**, **getAttributeNames** e **setAttribute** della interfaccia `ServletRequest`. Solo un valore può essere associato ad un nome di attributo.

6.4.3 Header HTTP

Un servlet può accedere agli header di una richiesta HTTP attraverso i metodi **getHeader**, **getHeaders** e **getHeaderNames** dell'interfaccia `HttpServletRequest`. Gli header potrebbero contenere **String** che rappresentano dati **int** o **Date**. Per accedere all'header in uno di questi formati, sono disponibili i metodi **getIntHeader** (viene generata un'eccezione **NumberFormatException** se il metodo non può tradurre il valore dell'header in int) e **getDateHeader** (viene generata una eccezione **IllegalArgumentException** se il metodo non può tradurre il valore dell'header in oggetto `Date`) dell'interfaccia `HttpServletRequest`.

6.4.4 I Cookies

L'interfaccia `HttpServletRequest` fornisce il metodo **getCookies** per ottenere un array di cookie presenti nella richiesta. Ma cosa sono i cookie? Alcuni siti Web memorizzano dati nel computer dell'utente in un piccolo file di testo, denominato appunto **cookie**.

I cookie contengono informazioni relative all'utente e alle preferenze espresse. Se ad esempio l'utente esegue una ricerca relativa all'orario di un volo in un sito di un'agenzia aerea, il sito Web può creare un cookie che contiene l'itinerario dell'utente. In alternativa, il cookie potrebbe contenere semplicemente una registrazione delle pagine visitate all'interno del sito, come ausilio utilizzato dal sito stesso per personalizzare la successiva visita da parte dell'utente.

Nei cookie possono essere memorizzate solo le informazioni fornite o le scelte effettuate dall'utente durante la visita del sito Web. Il sito non è ad esempio in grado di rilevare l'indirizzo di posta elettronica, a meno che l'utente non lo digiti. La creazione di un cookie da parte di un sito Web non comporta l'accesso di questo sito o di altri siti al resto del computer. Inoltre solo il sito che ha creato il cookie sarà in grado di leggerlo. Generalmente i browser Web sono configurati per consentire la creazione di cookie; tuttavia l'utente può configurare il suo browser per impedire l'accettazione di qualsiasi cookie.

Di seguito si esaminano i limiti del protocollo HTTP che portano all'utilizzo dei cookie e come questi possono essere manipolati tramite servlet.

6.4.4.1 Perché i cookie? I limiti del protocollo HTTP

Il limite maggiore del protocollo HTTP è legato alla sua natura di protocollo **non transazionale**, in altre parole il protocollo HTTP non è in grado di mantenere dati persistenti tra i vari pacchetti. Fortunatamente esistono due tecniche per aggirare il problema: i cookie e la gestione di sessioni utente.

I cookie sono, come detto, piccoli file contenenti un'informazione scritta all'interno secondo un certo formato, che vengono depositati dal server sul client e contengono informazioni specifiche relative all'applicazione che li genera. Se utilizzati correttamente consentono di memorizzare dati utili alla gestione del flusso di informazioni creando delle entità persistenti in grado di fornire un punto di appoggio per garantire un minimo di transazionalità all'applicazione Web. Formalmente i cookie contengono al loro interno una singola informazione nella forma **nome=valore** più una serie di informazioni aggiuntive che rappresentano:

- Il dominio applicativo del cookie
- Il path dell'applicazione
- La durata della validità del file
- Un valore booleano che identifica se il cookie è criptato o no

Il **dominio applicativo** del cookie consente al browser di determinare se, al momento di inviare una richiesta HTTP, dovrà associarle il cookie da inviare al server. Un valore come `www.myserver.it` indicherà al browser che il cookie sarà valido solo per la macchina `www` all'interno del dominio `myserver.it`.

Il **path dell'applicazione** rappresenta il percorso virtuale dell'applicazione per la quale il cookie è valido. Un valore del tipo `/` indica al browser che qualunque richiesta HTTP da inviare al dominio definito nel campo precedente dovrà essere associata al cookie. Un valore del tipo `/servlet/ServletProva` indicherà invece al browser che il cookie dovrà essere inviato in allegato a richieste HTTP del tipo `http://www.myserver.it/servlet/ServletProva`.

La terza proprietà, comunemente detta **expiration**, indica il tempo di durata della validità del cookie in secondi prima che il browser lo cancelli definitivamente. Mediante questo campo è possibile definire cookie persistenti, ossia senza data di scadenza.

6.4.4.2 Manipolare cookies con i Servlet

Un servlet può inviare uno o più cookie ad un browser mediante il metodo **addCookie** definito nell'interfaccia `HttpServletResponse`, il quale consente di appendere l'entità ad un messaggio di risposta al browser. Viceversa, i cookie associati ad una richiesta HTTP possono essere recuperati da un servlet utilizzando il metodo **getCookie** definito nell'interfaccia `HttpServletRequest`, che ritorna un array di oggetti.

Il cookie viene rappresentato in Java dalla classe **javax.servlet.http.Cookie**, il cui prototipo è riportato nel codice seguente.

```
package javax.servlet.http;
public class Cookie implements Cloneable {
    public Cookie(String name, String value); // è il costruttore della classe e prende due parametri di tipo
        // stringa che rappresentano il nome ed il valore che si desidera assegnare all'oggetto che si sta
    creando
    public String getComment( ); // restituisce una stringa contenente il valore dell'attributo commento
    public String getDomain( );
    public int getMaxAge( );
    public String getName( ); // restituisce il nome del cookie che viene impostato al momento della creazione e
        // non può più essere cambiato
    public String getPath( );
    public boolean getSecure( );
    public String getValue( ); // restituisce una stringa contenente il valore del cookie
    public int getVersion( );
    public void setComment(String purpose); // imposta l'attributo commento del cookie
    public void setDomain(String pattern);
    public void setMaxAge(int expiry); // imposta l'età massima raggiungibile dal cookie in secondi
```

```

public void setPath(String uri);
public void setSecure(boolean flag);
public void setValue(String newValue); // imposta un nuovo valore per il cookie
public void setVersion(int v);
}

```

Esempio 6.5 Prototipo della classe Cookie

6.4.4.3 Un esempio completo

Nell'esempio si implementa un servlet che alla prima chiamata invia al server una serie di cookie mentre per ogni chiamata successiva ne stampa semplicemente il valore contenuto.

```

import javax.servlet.* ;
import javax.servlet.http.* ;
import java.io.* ;
public class TestCookie extends HttpServlet {
    private int numrichiesta=0;
    public void service (HttpServletRequest req, HttpServletResponse res) throws ServletException, IOException {
        Cookie cookies = null;
        res.setContentType("text/html");
        PrintWriter out = res. getWriter ( );
        out.println("<html>");
        out.println("<head><title>Cookie Test</title></head>");
        out.println("<body>");
        switch(numrichiesta) {
            case 0:
                // appende 10 cookie alla risposta HTTP
                for (int i=0; i<10; i++) {
                    String nome="cookie"+i;
                    String valore="valore"+i;
                    cookies = new Cookie(nome,valore);
                    cookies.setMaxAge(1000);
                    res.addCookie(cookies);
                } // fine for
                out.println("<h1>I cookie sono stati appesi a questa risposta<h1>");
                numrichiesta++;
                break;
            default :
                // ricava l'array dei cookie e stampa le coppie nome=valore
                Cookie cookies[] = req.getCookies( );
                for (int j=0; j<cookies.length; j++) {
                    Cookie appo = cookies[j];
                    out.println("<h1>"+appo.getName( )+" = "+appo.getValue( )+"<h1>");
                } // fine for
            } // fine switch
        out.println("</body></html>");
        out.close( );
    } // fine metodo service
} // fine classe TestCookie

```

Esempio 6.6 TestCookie.java

6.4.5 Attributi SSL

Se una richiesta è stata trasmessa su un protocollo sicuro, come **HTTPS**, questa informazione può essere mostrata tramite il metodo **isSecure** dell'interfaccia ServletRequest. Il Web container deve mostrare gli attributi che descrivono l'algoritmo di cifratura utilizzato: **cipher_suite** indica la tecnica di cifratura, **key_size** la dimensione in bit della chiave di cifratura usata dall'algoritmo.

Attributo	Nome dell'attributo	Tipo Java
cipher suite	javax.servlet.requets.cipher_suite	String
bit size of the algorithm	javax.servlet.requets.key_size	Integer

Tabella 6.1 Attributi SSL

6.5 La Risposta

L'oggetto risposta incapsula tutte le informazioni che devono essere ritornate dal server al client. Nel protocollo HTTP tali informazioni sono trasmesse sia tramite gli header HTTP sia tramite il message body della richiesta.

L'oggetto risposta definisce il canale di comunicazione tra il servlet ed il client che ha inviato la richiesta. Questo oggetto mette a disposizione del servlet i metodi necessari per inviare al client le entità prodotte dalla manipolazione dei dati di input. Un'istanza dell'oggetto **ServletResponse** viene definita per implementazione della interfaccia **HttpServletResponse** definita all'interno del package **javax.servlet.http**, che definisce una specializzazione dell'interfaccia derivata rispetto al protocollo HTTP.

```
package javax.servlet;
import java.io.*;

public interface ServletResponse extends RequestDispatcher {
    public String getCharacterEncoding( );
    public ServletOutputStream getOutputStream( ) throws IOException;
    public PrintWriter getWriter( ) throws IOException;
    public void setContentLength(int length);
    public void .setContentType(String type);
}
```

Esempio 6.7 L'interfaccia ServletResponse

```
package javax.servlet.http;
import java.util.*;
import java.io.*;

public interface HttpServletResponse extends javax.servlet.ServletResponse {
    public static final int SC_CONTINUE = 100;
    public static final int SC_SWITCHING_PROTOCOLS = 101;
    public static final int SC_OK = 200;
    public static final int SC_CREATED = 201;
    public static final int SC_ACCEPTED = 202;
    public static final int SC_NON_AUTHORITATIVE_INFORMATION = 203;
    public static final int SC_NO_CONTENT = 204;
    public static final int SC_RESET_CONTENT = 205;
    public static final int SC_PARTIAL_CONTENT = 206;
    public static final int SC_MULTIPLE_CHOICES = 300;
    public static final int SC_MOVED_PERMANENTLY = 301;
    public static final int SC_MOVED_TEMPORARILY = 302;
    public static final int SC_SEE_OTHER = 303;
    public static final int SC_NOT_MODIFIED = 304;
    public static final int SC_USE_PROXY = 305;
    public static final int SC_BAD_REQUEST = 400;
    public static final int SC_UNAUTHORIZED = 401;
    public static final int SC_PAYMENT_REQUIRED = 402;
    public static final int SC_FORBIDDEN = 403;
    public static final int SC_NOT_FOUND = 404;
    public static final int SC_METHOD_NOT_ALLOWED = 405;
    public static final int SC_NOT_ACCEPTABLE = 406;
    public static final int SC_PROXY_AUTHENTICATION_REQUIRED = 407;
    public static final int SC_REQUEST_TIMEOUT = 408;
    public static final int SC_CONFLICT = 409;
```



```

public static final int SC_GONE = 410;
public static final int SC_LENGTH_REQUIRED = 411;
public static final int SC_PRECONDITION_FAILED = 412;
public static final int SC_REQUEST_ENTITY_TOO_LARGE = 413;
public static final int SC_REQUEST_URI_TOO_LONG = 414;
public static final int SC_UNSUPPORTED_MEDIA_TYPE = 415;
public static final int SC_INTERNAL_SERVER_ERROR = 500;
public static final int SC_NOT_IMPLEMENTED = 501;
public static final int SC_BAD_GATEWAY = 502;
public static final int SC_SERVICE_UNAVAILABLE = 503;
public static final int SC_GATEWAY_TIMEOUT = 504;
public static final int SC_HTTP_VERSION_NOT_SUPPORTED = 505;
void addCookie(Cookie cookie);
boolean containsHeader(String name);
String encodeRedirectUrl(String url); //deprecated
String encodeRedirectURL(String url);
String encodeUrl(String url);
String encodeURL(String url);
void sendError(int statusCode) throws java.io.IOException;
void sendError(int statusCode, String message) throws java.io.IOException;
void sendRedirect(String location) throws java.io.IOException;
void setDateHeader(String name, long date);
void setHeader(String name, String value);
void setIntHeader(String name, int value);
void setStatus(int statusCode);
void setStatus(int statusCode, String message);
}

```

Esempio 6.8 L'interfaccia HttpServletResponse

L'interfaccia HttpServletResponse definisce i metodi per impostare dati relativi all'entità inviata nella risposta (lunghezza e tipo MIME), fornisce informazioni relativamente al set di caratteri utilizzato dal browser (in HTTP questo valore viene trasmesso nell'header **Accept-Charset** del protocollo), infine fornisce al servlet l'accesso al canale di comunicazione per inviare l'entità al client. I due metodi deputati alla trasmissione sono definiti nell'interfaccia ServletResponse:

- **getOutputStream** restituisce un oggetto di tipo **ServletOutputStream** e consente di inviare dati al client in forma binaria (ad esempio il browser richiede il download di un file).
- **getWriter** restituisce un oggetto di tipo **PrintWriter** e quindi consente di trasmettere entità allo stesso modo di una **System.out**. Nel caso in cui sia necessario utilizzare il metodo **setContentype**, sarà obbligatorio effettuare la chiamata prima di utilizzare il canale di output.

6.5.1 Buffering

Ad un servlet container è permesso, ma non richiesto, di bufferizzare l'output per il client. Il buffering consente una maggiore efficienza. I server permettono ai servlet di accedere e specificare i parametri di buffering se il servlet sta usando un ServletOutputStream o un PrintWriter, attraverso i metodi **getBufferSize**, **setBufferSize**, **isCommitted**, **reset**, **resetBuffer** e **flushBuffer** dell'interfaccia ServletResponse.

6.5.2 Header HTTP

Un servlet può impostare gli header di una risposta HTTP tramite i metodi **setHeader**, **setIntHeader**, **setDateHeader**, **addHeader**, **addIntHeader** e **addDateHeader**, forniti dall'interfaccia HttpServletResponse.

6.5.3 Notificare errori utilizzando Servlet

Un **codice di stato** è un intero a tre cifre ed è il risultato dell'elaborazione da parte del server della richiesta. La prima cifra di un codice di stato definisce la regola per suddividere i vari codici:

- **1xx** : Informativo - Richiesta ricevuta, continua l'elaborazione dei dati;
- **2xx** : Successo - L'azione è stata ricevuta con successo, compresa ed accettata;
- **3xx** : Ridirezione - Ulteriori azioni devono essere compiute al fine di completare la richiesta;
- **4xx** : Client-Error - La richiesta è sintatticamente errata e non può essere soddisfatta;
- **5xx** : Server-Error - Il server non ha soddisfatto la richiesta apparentemente valida.

All'interno dell'interfaccia `HttpServletResponse`, oltre ai prototipi dei metodi vengono dichiarate tutta una serie di costanti che rappresentano appunto i codici di stato (o codici di errore) come definiti dallo standard HTTP. Il valore di queste variabili costanti può essere utilizzato con i metodi **`sendError`** e **`setStatus`** per inviare al browser particolari messaggi con relativi codici di errore. Un esempio è realizzato nel servlet seguente.

```
import javax.servlet.* ;
import javax.servlet.http.* ;
import java.io ;
public class TestStatus extends HttpServlet {
    public void service (HttpServletRequest req, HttpServletResponse res) throws ServletException, IOException {
        res.setContentType("text/html");
        ServletOutputStream out = res.getOutputStream();
        out.println("<html>");
        out.println("<head><title>Test Status Code</title></head>");
        out.println("<body>");
        res.sendError(res.SC_OK, "Il sito è stato temporaneamente sospeso");
        out.println("</body></html>");
    }
}
```

Esempio 6.9 `TestStatus.java`

6.5.4 Chiusura di un oggetto Response

Quando una risposta è terminata, il container deve immediatamente inviare (flush) al client tutto il contenuto rimanente nel buffer. I seguenti eventi indicano che il servlet ha soddisfatto la richiesta e che l'oggetto response può essere chiuso:

- ✘ E' terminato il metodo `service` del servlet.
- ✘ La quantità di contenuto specificata nel metodo **`setContentLength`** della risposta è stata scritta sulla risposta.
- ✘ E' chiamato il metodo **`sendError`**.
- ✘ E' chiamato il metodo **`sendRedirect`**.

6.6 Sessioni

L'HTTP è per sua natura un protocollo stateless. Per costruire applicazioni Web efficaci, è imperativo che le richieste provenienti da un particolare client siano fra loro associate. Molte strategie per il **session tracking** si sono evolute nel tempo, ma sono tutte difficili e problematiche da usare direttamente per il programmatore.

Le specifiche 2.3 della tecnologia Servlet definiscono una semplice interfaccia **`HttpSession`** che permette ad un servlet container di usare ciascuno dei diversi approcci proposti per tener traccia di una sessione utente senza richiedere allo sviluppatore la conoscenza delle sfumature di ciascun approccio. Ogni sessione è associata quindi ad un oggetto **`HttpSession`**, memorizzato sul Web server, che contiene tutte le informazioni della sessione. Dopo essere stata creata, una sessione rimane disponibile fin quando non viene eliminata tramite esplicita richiesta di un servlet (o allo scadere di un timeout predefinito).

Una stessa sessione può essere condivisa da più servlet che collaborano tra loro, in modo che ognuno di essi possa tenere nota dei servizi già richiesti dallo stesso utente e operare di conseguenza. Per evitare interferenze tra servlet indipendenti, viene imposto che servlet indipendenti non possano utilizzare lo stesso nome per salvare la sessione.

6.6.1 Meccanismi per il Session Tracking

1. **Cookies.** Questo è il meccanismo più usato per il session tracking e deve essere supportato da tutti i servlet container. Il container manda un cookie al client. Questo ritornerà il cookie ad ogni successiva richiesta al server, associando senza ambiguità la richiesta con una sessione. Il nome del session tracking cookie deve essere **JSESSIONID**.
2. **Sessioni SSL.** Secure Socket Layer, la tecnologia di encryption (cifratura) usata nell'HTTPS, ha un meccanismo interno che permette a richieste multiple provenienti da un client di essere identificate senza ambiguità come facenti parte di una sessione. Un servlet container può usare facilmente questo meccanismo per definire una sessione.
3. **URL Rewriting.** E' il minimo comun denominatore del session tracking. Quando un client non accetta un cookie, tale meccanismo può essere usato dal server come base per il session tracking. L'URL Rewriting implica l'aggiunta di un dato, un session id, al cammino URL che è interpretato dal container per associare la richiesta ad una sessione. Il session id deve essere codificato come un parametro nella stringa URL. Il nome del parametro deve essere **jsessionId** (es. <http://www.myserver.it/catalog/index.html;jsessionId=1234>). Dal momento che i Web container devono supportare le sessioni HTTP anche quando il client non supporta l'uso e la creazione di cookie, essi comunemente supportano il meccanismo di URL Rewriting.

6.6.2 Cookie vs. Sessioni Utente

I cookie non sono uno strumento completo per memorizzare lo stato di un'applicazione Web. Di fatto sono spesso considerati dall'utente poco sicuri e pertanto non accettati dal browser, che li rifiuterà al momento dell'invio con un messaggio di risposta. Come se non bastasse il cookie ha un tempo massimo di durata prima di essere cancellato dalla macchina dell'utente.

I servlet risolvono questo problema permettendo di racchiudere l'attività di un client per tutta la sua durata all'interno di **sessioni utente**. Un servlet può utilizzare una sessione utente per memorizzare dati persistenti e dati specifici relativi all'applicazione, e recuperarli in qualsiasi istante sia necessario. Questi dati possono essere inviati al client all'interno dell'entità prodotta.

Ogni volta che un client effettua una richiesta HTTP, se in precedenza è stata definita una sessione utente legata ad esso, il servlet container identifica il client e determina la sessione a questo associata. Nel caso in cui il servlet lo richieda, il container gli mette a disposizione tutti gli strumenti per utilizzarla. Ogni sessione creata dal container è associata in modo univoco ad un identificativo (ID). Due sessioni non possono essere associate ad uno stesso ID.

6.6.3 Sessioni dal punto di vista di un Servlet

Il servlet container contiene le istanze delle sessioni utente rendendole disponibili ad ogni servlet che ne faccia richiesta (Figura 6.7 pagina seguente). Ricevendo un identificativo dal client, un servlet può accedere alla sessione a questo associata. Esistono molti modi per consentire al client di impostare l'identificativo di una sessione.

Tipicamente viene utilizzato il meccanismo dei **cookie persistenti**. Ogni volta che un client esegue una richiesta HTTP, il cookie contenente l'identificativo della richiesta viene trasmesso al server che ne ricava il valore contenuto e lo mette a disposizione dei servlet. Nel caso in cui il browser non consenta l'utilizzo di cookie esistono tecniche alternative che risolvono il problema.

In genere una sessione utente deve essere avviata da un servlet dopo aver verificato se già non ne esista una, utilizzando il metodo **getSession** di `HttpServletRequest`. Se il valore booleano passato a tale metodo vale `true` ed esiste già una sessione associata all'utente il metodo ritornerà semplicemente un oggetto che la rappresenta, altrimenti ne creerà una, ritornandola come oggetto di ritorno del metodo. Al contrario se il parametro di input vale `false`, il metodo tornerà la sessione se già esistente, null altrimenti. Quando un servlet crea una nuova sessione, il container

genera automaticamente l'identificativo ed appende un cookie contenente l'ID alla risposta HTTP in modo del tutto trasparente al servlet.

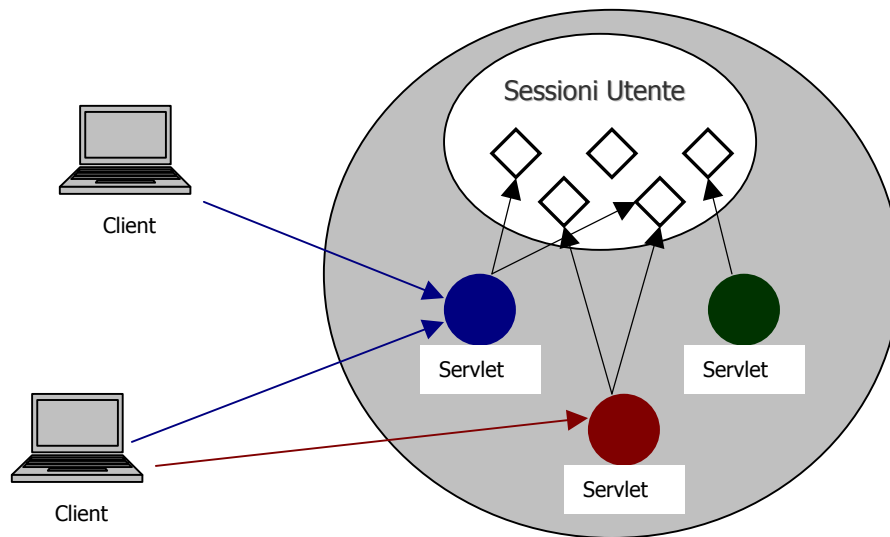


Figura 6.7 Le sessioni sono accessibili da tutti i servlet

6.6.4 La classe HttpSession

Un oggetto di tipo **HttpSession** viene restituito al servlet come parametro di ritorno del metodo `getSession`. Tale oggetto è definito dall'interfaccia **javax.servlet.http.HttpSession**.

```
package javax.servlet.http;
public interface HttpSession {
    long getCreationTime( );
    String getId( ); // restituisce una stringa contenente il nome della sessione
                       // il nome della sessione è unico e viene impostato da HttpSessionContext.
    long getLastAccessedTime( );
    int getMaxInactiveInterval( );
    HttpSessionContext getSessionContext( );
    Object getValue(String name); // restituisce l'oggetto collegato alla sessione identificata dal parametro name
                                   // (oppure null se non esiste un oggetto collegato alla sessione indicata)
    String[] getValueNames( );
    void invalidate( ); // invalida la sessione
    boolean isNew( );
    void putValue(String name, Object value); // collega l'oggetto value alla sessione identificata tramite il
                                                // parametro name. Eventuali oggetti precedentemente collegati vengono automaticamente rimpiazzati
    void removeValue(String name); // rimuove, se esiste, l'oggetto collegato alla sessione identificata da name
    int setMaxInactiveInterval(int interval);
}
```

Esempio 6.10 L'interfaccia HttpSession

Utilizzando i metodi **getId**, **getCreationTime**, **getLastAccessedTime**, **getMaxInactiveInterval** e **isNew** si possono ottenere le meta informazioni relative alla sessione che stiamo manipolando. È importante notare che i metodi che ritornano un valore che rappresenta un tempo rappresentano il dato in secondi. Sarà quindi indispensabile operare le necessarie conversioni per determinare informazioni tipo date ed ore. Il significato dei metodi sopra citati risulta chiaro leggendo il nome del metodo. Il primo ritorna l'identificativo della sessione, il secondo il tempo in secondi trascorso dalla creazione della sessione prima della richiesta corrente, il terzo il tempo in secondi trascorso dall'ultimo accesso alla sessione, infine il quarto l'intervallo massimo di tempo di inattività della sessione.

Quando creiamo una sessione utente, l'oggetto generato verrà messo in uno stato di **new** ad indicare che la sessione è stata creata ma non è attiva. Da un punto di vista puramente formale è ovvio che per essere attiva la sessione deve essere accettata dal client, ovvero una sessione viene accettata dal client solo nel momento in cui quest'ultimo invia al server per la prima volta l'identificativo della sessione. Il metodo **isNew** restituisce un valore booleano che indica lo stato della sessione.

I metodi **putValue**, **removeValue**, **getValueNames** e **getValue** consentono di memorizzare o rimuovere oggetti nella forma di coppie nome=oggetto all'interno della sessione. Consentono ad un servlet di memorizzare dati (in forma di oggetti) all'interno della sessione per poi utilizzarli ad ogni richiesta HTTP da parte dell'utente associato alla sessione. Questi oggetti saranno inoltre accessibili ad ogni servlet che utilizzi la stessa sessione utente: potranno essere recuperati conoscendone il nome associato.

6.6.5 Un esempio di gestione di una Sessione Utente

L'esempio che segue è relativo ad un servlet che conta il numero di accessi che un determinato utente ha effettuato sul sistema utilizzando il meccanismo delle sessioni.

```
import javax.servlet.* ;
import javax.servlet.http.* ;
import java.io.* ;
public class TestSession extends HttpServlet {
    public void service (HttpServletRequest req, HttpServletResponse res) throws ServletException, IOException {
        res.setContentType("text/html");
        PrintWriter out = res.getWriter ( );
        out.println("<html>");
        out.println("<head><title>Test di una sessione servlet</title></head>");
        out.println("<body>");
        HttpSession sessione = req.getSession(true);
        if(sessione.isNew( )) {
            out.println("<strong>Id della sessione: </strong>" +session.getId()+"<br>");
            out.println("<strong>Creata al tempo: </strong>" +session.creationTime()+"<br>");
            out.println("<strong>Questa è la tua prima connessione al server </strong>");
            session.putValue("ACCESSI", new Integer(1));
        } // fine if
        else {
            int accessi = ((Integer) session.getValue("ACCESSI")).intValue( );
            accessi++;
            session.putValue("ACCESSI", new Integer(accessi));
            out.println("<strong>Questa è la tua connessione numero: </strong>" +accessi);
        } // fine else
        out.println("</body></html>");
        out.close();
    } // fine service
} // fine TestSession
```

Esempio 6.11 TestSession.java

6.6.6 Durata di una Sessione Utente

Una sessione utente rappresenta un oggetto transiente la cui durata deve essere limitata al periodo di attività dell'utente sul server. Utilizzando i metodi **invalidate** e **setMaxInactiveInterval** è possibile invalidare una sessione o disporre che una volta superato l'intervallo massimo di inattività la sessione venga automaticamente resa inattiva dal container.

Nel protocollo HTTP, non c'è nessun segnale esplicito di terminazione quando un client non è più attivo. Ciò significa che il solo meccanismo che può essere usato per indicare quando un client non è più attivo è un intervallo di timeout. L'intervallo di timeout di default per le sessioni è definito dal servlet container, e può essere ottenuto tramite il metodo **setMaxInactiveInterval** dell'interfaccia **HttpSession**. Questo timeout può essere cambiato usando, come detto, il metodo

setMaxInactiveInterval. Gli intervalli sono definiti in secondi. Per definizione, se il timeout period per una sessione è settato a -1, la sessione non scadrà mai.

6.6.7 URL Rewriting

Il browser ha la facoltà di accettare o no cookie da parte di un server Web. Quando la ricezione dei cookie è disabilitata è impossibile per il server tracciare l'identificativo della sessione e, di conseguenza, permettere ad un servlet di accederle. Un metodo alternativo consiste nel codificare l'identificativo della sessione all'interno dell'URL che il browser invia al server all'interno di una richiesta HTTP.

Questo meccanismo deve necessariamente essere supportato dal server, che dovrà prevedere l'utilizzo di caratteri speciali all'interno dell'URL. Server differenti potrebbero utilizzare metodi differenti. L'interfaccia HttpServletResponse mette a disposizione il metodo **encodeURL** che accetta come parametro di input un URL, determina se è necessario riscriverlo ed eventualmente codifica all'interno dell'URL l'identificativo della sessione.

Capitolo 7 La tecnologia JavaServer Pages

La tecnologia JavaServer Pages fu presentata in occasione di **Java One** del 1998, una manifestazione organizzata dalla Sun Microsystems a San Francisco, per la presentazione delle innovazioni legate a Java. Passata in un primo tempo pressoché inosservata dalla comunità di sviluppatori Web (principalmente a causa dell'iniziale mancanza di software per la prova e lo sviluppo delle applicazioni), JSP oggi, nonostante la sua giovane età, è considerata dalla maggior parte degli addetti ai lavori come una tecnologia competitiva che giocherà un ruolo fondamentale nell'evoluzione del Web futuro.



Figura 7.1 Il Logo della tecnologia JSP

JavaServer Pages è una tecnologia **server-side** promossa dalla piattaforma J2EE per realizzare pagine Web dai **contenuti dinamici**, come HTML, DHTML, XHTML e XML. Tale tecnologia permette la creazione di pagine Web con contenuti dinamici con la massima potenza e flessibilità. Costruito utilizzando la sintassi Java, JSP è un linguaggio multiplatforma, interamente interpretato dal server e flessibile. Le pagine JSP sono un'evoluzione dei già collaudati servlet Java, e sono state create per separare i contenuti dalla loro presentazione: una pagina JSP si presenta, infatti, come un normale documento in linguaggio HTML, frammentato da sezioni di codice Java. Si potranno quindi modificare le parti sviluppate in Java lasciando inalterata la struttura HTML o viceversa.

La versione più recente delle specifiche JSP è la **1.2**. Tali specifiche sono state sviluppate dell'Expert Group JSR053 sotto il Java Community Process (info su <http://jcp.org/jsr/detail/53.jsp>).

JSP home page	http://java.sun.com/products/jsp
Servlet home page	http://java.sun.com/products/servlet
Java 2 Platform, Standard Edition	http://java.sun.com/products/jdk/1.3
Java 2 Platform, Enterprise Edition	http://java.sun.com/j2ee
XML in the Java Platform home page	http://java.sun.com/xml
JavaBeans™ technology home page	http://java.sun.com/beans
XML home page at W3C	http://www.w3.org/XML
HTML home page at W3C	http://www.w3.org/MarkUp
XML.org home page	http://www.xml.org

Tabella 7.1 Alcuni siti Web collegati

7.1 Overview

7.1.1 Concetti generali

JSP consente la creazione di pagine Web a contenuto dinamico: la pagina HTML (di per se statica nel suo contenuto) è prodotta in modo dinamico dal server Web. Quest'ultimo, mediante l'interpretazione di determinati comandi, costruisce una pagina HTML personalizzata (in relazione a determinate condizioni, eventi o richieste dell'utente) ed eventualmente sempre diversa. Il browser (lato client) riceverà la pagina HTML generata dal server come una normale pagina Web: non sono richieste particolari funzionalità aggiuntive al browser, il quale deve semplicemente essere in grado di interpretare HTML standard.

Questo obiettivo può essere raggiunto per mezzo di molte tecnologie e non a caso diversi sono gli strumenti di programmazione esistenti al fine di generare pagine Web dinamiche. Le principali soluzioni, o per lo meno le più note, sono gli script CGI, le pagine ASP, le pagine PHP (acronimo di Pre Hypertext Processor) e, appunto, le pagine JSP.

7.1.2 Che cosa è una pagina JSP?

Una pagina JSP è un documento **text-based** che descrive come processare una richiesta per creare una risposta dinamica. Le caratteristiche chiave sono:

- Direttive standard
- Action standard
- Elementi di scripting
- Tag Library
- Template Data

7.1.3 Vantaggi di JSP

In primo luogo (a differenza di soluzioni come ASP e PHP) le JSP sono in chiave Java e quindi eredi di tutti gli indiscussi vantaggi che hanno reso questo linguaggio di programmazione uno dei più sfruttati nel mondo della programmazione ad oggetti; prima fra tutti la quasi totale **portabilità** (intesa come capacità di un software di funzionare correttamente su stazioni che utilizzano un qualsiasi tipo di piattaforma e sintetizzata dallo slogan Write Once Run Anywhere). Le pagine JSP sono un'ottima soluzione per l'implementazione di programmi applicativi attivi dal lato server; essendo svincolate dal tipo di sistema operativo non necessitano di nessuna modifica nemmeno nel passaggio tra server gestiti in maniera diversa. Ovviamente l'aspetto della portabilità non deve essere visto esclusivamente sotto l'aspetto server: anche dal lato client questa tecnologia offre un completo svincolamento sia dal tipo di sistema operativo sia dal tipo di browser: Java viene infatti interpretato dal server sollevando il browser da questo compito, e limitando la sua funzione a semplice interprete di pagine HTML.

Altro aspetto importante è la **separazione dei ruoli**. La possibilità di fondere HTML con Java senza che nessuno interferisca con l'altro consente di isolare la rappresentazione dei contenuti dinamici dalle logiche di presentazione. Il designer potrà concentrarsi solo sull'impaginazione dei contenuti, i quali saranno inseriti dal programmatore che a sua volta non dovrà preoccuparsi dell'aspetto puramente grafico.

JSP permette poi il **riutilizzo del codice** (incapsulamento di funzionalità): si possono infatti includere nelle pagine JSP componenti JavaBeans ed EJBs e tag library che rendono molto più abbordabile l'implementazione di applicazioni anche complesse, e molto più semplice la modifica e la manutenzione delle pagine stesse.

La tecnologia JSP infine ha caratteristiche che permettono la creazione di buoni strumenti di authoring: disporre di buoni tool porta ad un miglioramento della produttività.

7.1.4 Applicazione Web

Il concetto di applicazione Web, unitamente a quelli di ServletContext, sessioni, richieste e risposte, è ereditato dalle Servlet specification. Un'applicazione Web può essere composta da:

- ✗ Ambiente Java Runtime che gira sul server (richiesto)
- ✗ Pagine JSP e Servlet che gestiscono richieste e generano contenuti dinamici
- ✗ Componenti server-side JavaBeans che incapsulano il comportamento e lo stato
- ✗ HTML statico, DHTML, XHTML, XML e pagine simili
- ✗ Applet Java, componenti JavaBeans e file class Java client-side
- ✗ Ambienti Java Runtime che girano sui client (scaricabili tramite il Plugin e la tecnologia Java Web Start)

7.1.5 Confronto con altre tecnologie

In questa sezione si propone un confronto fra la tecnologia JSP e altre tecniche usate per la generazione di contenuti dinamici lato server.

Rispetto a CGI, due sono le differenze sostanziali. Primo, le pagine JSP possono mantenere lo stato sul server fra le richieste (tramite le sessioni). Secondo, una pagina JSP viene compilata solo alla prima richiesta e non deve essere caricata ogni volta, mentre nel caso di CGI, nella implementazione più semplice, ogni richiesta crea un nuovo processo, questo viene caricato,

eseguito e stoppato una volta che la richiesta è stata soddisfatta: ciò comporta, evidentemente, un impiego notevole delle risorse messe a disposizione dal server.

La principale tecnologia concorrente di JSP è ASP di Microsoft. Le tecnologie sono abbastanza simili nel modo in cui esse supportano la creazione di pagine dinamiche usando template HTML, codice di scripting e componenti per la business logic. La tabella seguente propone un paragone fra le due tecnologie.

Caratteristiche	JSP	ASP
Piattaforme	Tutte le maggiori piattaforme	Microsoft
Linguaggio Base	Java	VBScript o JScript
Componenti	Tag JSP, JavaBeans, o EJB	COM/DCOM
Compilazione/Interpretazione	Una volta	Ogni istanza

Tabella 7.2 Confronto tra JSP e ASP

Le pagine JSP sono interpretate una sola volta in Java bytecode, e reinterpretate solo quando il file viene modificato. Inoltre JSP gira su tutte le maggiori piattaforme e fornisce, rispetto ad ASP, migliori strumenti per una più netta separazione fra il codice ed il template HTML, attraverso JavaBeans, Enterprise JavaBeans e le tag library (librerie di tag).

7.1.6 JavaServer Pages vs. Servlet

I servlet sono uno degli elementi fondamentali da cui deriva e su cui si appoggia la tecnologia JSP. Un servlet non è altro che un'applicazione Java in esecuzione su una JVM residente su un server. Questa applicazione tipicamente esegue una serie di operazioni che producono in output un codice HTML che verrà poi inviato direttamente al client per venire interpretato da un qualsiasi browser.

Come già descritto nel capitolo precedente, i servlet offrono la possibilità di costruire pagine Web dal contenuto dinamico usando linguaggio Java e sono supportati dai maggior Web server. Il comportamento di un servlet richiamato in una pagina HTML può, in effetti, rendere possibile la generazione di pagine dinamiche; per questo motivo è lecito chiedersi quale sia il valore aggiunto della tecnologia JavaServer Pages.

Se dal punto di vista del programmatore una pagina JSP è un documento di testo contenente tag HTML e codice Java, dal punto di vista del server una pagina JSP è utilizzata allo stesso modo di un servlet: è tradotta e compilata in una classe servlet di tipo `HttpServlet` (chiamata **JSP page implementation class**) che viene istanziata al momento della richiesta e che gestisce le richieste e crea le risposte. Tipicamente il Web server memorizza su disco tutte le definizioni di classe ottenute dal processo di compilazione per riutilizzare il codice già compilato. L'unica volta che una pagina JSP viene compilata è al momento del suo primo accesso da parte di un client o dopo modifiche apportate dal programmatore: il client accederà sempre all'ultima versione prodotta.

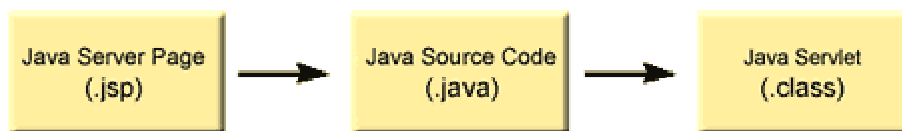


Figura 7.2 Traduzione e compilazione di una pagina JSP

Quando ad un Web server (che supporta JSP) è richiesta una pagina JSP, esso verifica, innanzi tutto, se tale pagina è già stata compilata (viceversa produce il relativo bytecode), quindi carica ed esegue il codice Java della pagina JSP come un qualsiasi altro servlet, producendo in output la pagina HTML da inviare al browser. Il grande vantaggio che se ne ottiene è che, in questo caso, la presenza di elementi di programmazione differenti (HTML, EJB, servlet) permette di diversificare i vari compiti, semplificando la loro realizzazione.

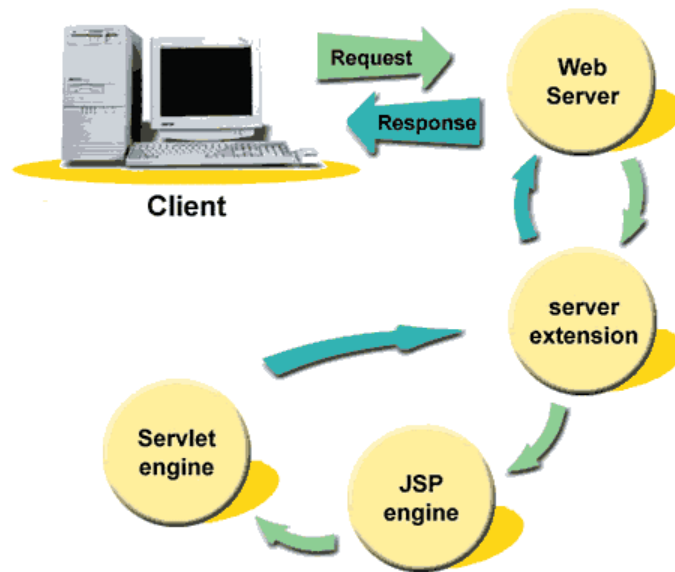


Figura 7.3 Elaborazione di una richiesta

Rispetto all'invocazione di servlet puri per la generazione di pagine Web dinamiche, si semplifica la stesura e la modifica dell'HTML poiché è possibile separare la sezione di produzione dei contenuti da quella di visualizzazione vera e propria. I vantaggi di questo disaccoppiamento si ripercuotono anche sulla modalità di sviluppo e manutenzione dell'applicazione, permettendo a team di lavoro differenti di lavorare in maniera indipendente ma parallela.

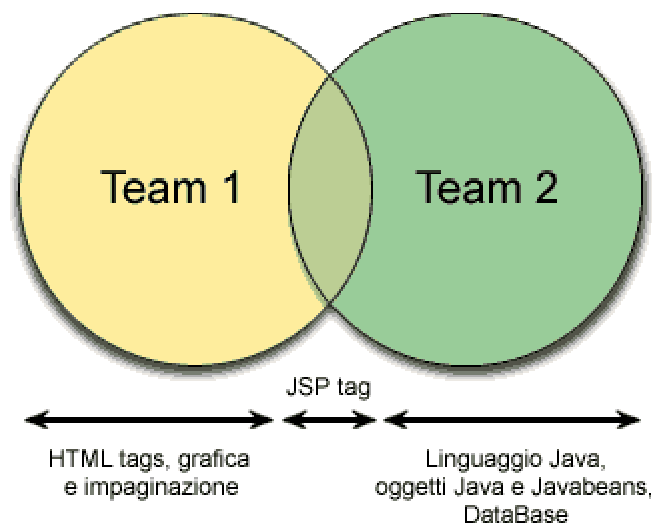


Figura 7.4 Separazione dei ruoli

7.1.7 Un primo esempio

L'esempio che segue riguarda una pagina JSP che visualizza nel browser il messaggio "PRIMA PROVA JSP".

```
<!-- PROVA.JSP -->
<HTML>
<BODY>
<% out.println("PRIMA PROVA JSP"); %>
</BODY>
</HTML>
```

Esempio 7.1 prova.jsp

Come si può notare, le sezioni racchiuse tra i tag `<%` e `%>` sono quelle che contengono le istruzioni in linguaggio Java; gli stessi tag sono utilizzati anche nelle pagine ASP. Per essere riconosciuti dal browser come pagine JSP i file devono essere salvati con estensione **.jsp**. La prima volta che si effettua la richiesta di visualizzazione del file, quest'ultimo viene compilato, creando un oggetto servlet, che sarà archiviato in memoria per servire le richieste successive; solo dopo questi passaggi l'output è mandato al browser, che potrà interpretarlo come fosse una semplice pagina HTML. Ad ogni richiesta successiva il server controlla in primo luogo se sulla pagina JSP è stata effettuata qualche modifica: in caso negativo richiama il servlet già compilato, altrimenti esegue nuovamente la compilazione e memorizza il nuovo servlet da richiamare.

Considerando il file `prova.jsp`, il servlet generato sarà costituito dal file Java presentato di seguito (il quale varia leggermente in base al motore JSP utilizzato).

```
import javax.servlet.*;
import javax.servlet.http.*;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import allaire.jrun.jsp.JRunJSPStaticHelpers;
public class jrun_prova2ejspa extends allaire.jrun.jsp.HttpJSPServlet implements allaire.jrun.jsp.JRunJspPage {
    private ServletConfig config;
    private ServletContext application;
    private Object page = this;
    private JspFactory _jspFactory = JspFactory.getDefaultFactory( );
    public void _jspService(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, java.io.IOException {
        if(config == null) {
            config = getServletConfig();
            application = config.getServletContext( );
        } // fine if
        response.setContentType("text/html; charset=ISO-8859-1");
        PageContext pageContext = _jspFactory.getPageContext(this,request, response, null, true, 8192, true);
        JspWriter out = pageContext.getOut( );
        HttpSession session = pageContext.getSession( );
        try {
            out.print("<!-- PROVA.JSP -->\r\n\r\n<html>\r\n\r\n<body>\r\n\r\n");
            out.println ("PRIMA PROVA JSP");
            out.print("\r\n\r\n</body>\r\n\r\n</html>");
        } catch(Throwable t) {
            if(t instanceof ServletException) throw (ServletException) t;
            if(t instanceof java.io.IOException) throw (java.io.IOException) t;
            if(t instanceof RuntimeException) throw (RuntimeException) t;
            throw JRunJSPStaticHelpers.handleException(t, pageContext);
        } finally {
            _jspFactory.releasePageContext(pageContext);
        }
    }
    private static final String[] _dependencies_ = {"/prova.jsp",null};
    private static final long[] _times_ = {988194570000L,0L};
    public String[] _getDependencies() {
        return _dependencies_;
    }
    public long[] _getLastModifiedTimes() {
        return _times_;
    }
    public int _getTranslationVersion(){
        return 14;
    }
}
```

Esempio 7.2 La JSP page implementation class di `prova.jsp`

Analizzando il codice riportato si può notare come il metodo che effettua il servizio vero e proprio è il metodo `_jspService`. Esso in un primo tempo verifica la configurazione del server (se non è stata effettuata se ne occupa), per poi impostare in maniera adeguata i parametri che permettono la comunicazione dell'output voluto (viene per esempio specificato che il testo in uscita è codice HTML). Effettuato il controllo della configurazione vengono eseguite le istruzioni per cui il servlet è stato creato (in questo caso la stampa a video del messaggio), tenendo in considerazione gli eventuali errori e creando delle adeguate eccezioni. Con l'ultima istruzione, il servlet viene rilasciato. Gli altri metodi servono per fornire alcune informazioni riguardanti il servlet: quando è stato modificato l'ultima volta, la versione della traduzione, il nome del file contenente la pagina JSP cui fa riferimento.

7.2 Elementi di una pagina JSP

7.2.1 Direttive

Le direttive sono istruzioni che avvertono il JSP container di una particolare impostazione da utilizzare. Sono processate quando la pagina JSP viene compilata. Specificano le caratteristiche globali della pagina (per esempio le informazioni relative al tipo di compilazione da effettuare o l'eventuale utilizzo di tag personalizzati) e non producono output per il client. Tutte le direttive hanno come scope la pagina JSP. La sintassi per le direttive è la seguente:

```
<%@ directive { attr="value" }* %>
```

Al momento sono definite tre direttive: `page`, `include` e `taglib`.

7.2.1.1 La direttiva page

La direttiva `page` definisce una serie di proprietà che vengono applicate all'intera pagina JSP attraverso la modifica di specifici attributi. Una pagina JSP può contenere diverse direttive `page`, eccetto che per l'attributo `import`. La sintassi è la seguente:

```
<%@ page page_directive_attr_list %>
```

La `page_directive_attr_list` è una lista di attributi, opportunamente valorizzati, descritti di seguito.

- **language="scriptingLanguage"** dichiara al server il linguaggio utilizzato all'interno della pagina JSP. In JSP 1.2 l'unico linguaggio di scripting definito e richiesto è "java".
- **extends="className"** definisce la classe base a partire dalla quale viene definito il servlet al momento della compilazione. Generalmente non viene utilizzato.
- **import="importList"** è simile all'istruzione `import` di una definizione di classe Java; deve essere una delle prime direttive e comunque comparire prima di altri tag JSP. Alcune classi sono importate di default: **java.lang.***, **javax.servlet.***, **javax.servlet.jsp.*** e **javax.servlet.http.***.
- **session="true | false"** di default vale `true` e significa che la pagina richiede la partecipazione in una sessione HTTP, e quindi i dati appartenenti alla sessione utente sono disponibili dalla pagina JSP.
- **buffer="none | 8kb | sizekb"** determina se l'output stream della pagina JSP utilizza o no un buffer di scrittura dei dati. Di default la dimensione è di 8k. Questa direttiva va utilizzata accostata alla direttiva **autoflush**.
- **autoFlush="true | false"** se impostato a `true` (default) svuota il buffer di output quando risulta pieno invece di generare una eccezione. Non è corretto impostare `autoFlush="false"` se anche `buffer="none"`.
- **isThreadSafe="true | false"** indica se la pagina può servire più richieste contemporaneamente. Di default è impostato a `true` e indica all'ambiente che il programmatore si preoccuperà di gestire gli accessi concorrenti mediante blocchi sincronizzati. Se impostato a `false` viene utilizzata di default l'interfaccia **SingleThreadModel** in fase di compilazione.

- **info="info_text"** fornisce informazioni sulla pagina che si sta accedendo attraverso il metodo **Servlet.getServletInfo()**.
- **errorPage="error_URL"** fornisce il path della pagina JSP che verrà richiamata in automatico per gestire eccezioni che non vengono controllate all'interno della pagina attuale.
- **isErrorPage="true | false"** definisce la pagina come una pagina di errore.
- **contentType="ctInfo"** definisce il tipo MIME della pagina prodotta dalla esecuzione.
- **pageEncoding="peInfo"** definisce la codifica dei caratteri per la pagina JSP.

Di seguito si propone un esempio di utilizzo della direttiva page.

```
<%@ page language="java" import="java.sql.*" %>
<%@ page buffer="5kb" autoFlush="false" %>
<%@ page errorPage="error.jsp" %>
```

Esempio 7.3 Utilizzo della direttiva page

7.2.1.2 La direttiva include

La direttiva include serve ad includere un file di testo o di codice nella pagina JSP corrente ed al punto specifico. Il file incluso non dovrebbe essere un'altra pagina dinamica. La sintassi è la seguente:

```
<%@ include file="relativeURL" %>
```

```
<HTML>
<BODY bgcolor="white">
<font color="blue">
The current date and time are
<%@ include file="date.jsp" %>
</font>
</BODY>
</HTML>
```

Esempio 7.4 Utilizzo della direttiva include

7.2.1.3 La direttiva taglib

La direttiva taglib permette di utilizzare una libreria di tag personalizzati (tag library). Tale direttiva dichiara che la pagina usa una tag library. La sintassi è la seguente:

```
<%@ taglib uri="tagLibraryURI" prefix="tagPrefix" %>
```

dove l'**uri** indica il nome dell'archivio della libreria dei tag che si intende utilizzare e **prefix** è il prefisso da utilizzare per richiamare il tag (<prefix>:<tagname>).


Di seguito viene proposto un esempio di utilizzo di tag library.

```
<%@ taglib uri="http://www.myserver.it/supertags" prefix="super" />
...
<super:doMagic>
...
</super:doMagic>
```

Esempio 7.5 Utilizzo della direttiva taglib


7.2.2 Elementi di scripting

Quando si progetta un sito Web con contenuti interattivi e dinamici, potrebbe essere utile usare piccole porzioni di codice per la generazione dei contenuti. Gli elementi di scripting sono appunto usati per includere codice Java in pagine JSP. Permettono di dichiarare variabili e metodi, di includere codice arbitrario e valutare espressioni. Sono di tre tipi: dichiarazioni (<%!....%>), scriptlet (<%....%>) ed espressioni (<%=....%>). E' da sottolineare comunque che per rendere il codice più facile da leggere e mantenere, le pagine JSP dovrebbero essere usate principalmente per la presentazione. Anche se la gran parte di una applicazione potrebbe essere sviluppata con la tecnologia JSP, la presenza di grandi quantità di codice nelle pagine JSP le rende difficili da aggiornare.

 Le **dichiarazioni** permettono la definizione di variabili e metodi globali rispetto alla pagina. Sono incluse all'interno della coppia di tag <%! declaration %>. Le dichiarazioni vengono inizializzate quando viene inizializzata una pagina JSP ed hanno scope class. Una dichiarazione rappresenta dal punto di vista del Web server che compila la pagina JSP il blocco di dichiarazione dei dati membro o dei metodi della classe servlet generata.


```
<%// dichiarazione di una stringa %>
<%! String stringa=new string("ciao a tutti") %>
<% // dichiarazione di una funzione che dati due numeri in ingresso restituisce la loro somma %>
<%! public int somma (int primo, int secondo){
    return (primo + secondo);
} //somma
%>
```

Esempio 7.6 Dichiarazioni di una stringa e di una funzione in una pagina JSP

 Gli **scriptlet** sono piccoli frammenti di codice scripting il cui linguaggio è definito dal parametro language nella direttiva JSP page. Uno scriptlet deve essere inserito all'interno della coppia di tag <% scriptlet %>. Il codice qui contenuto può accedere a qualsiasi variabile o bean dichiarato e viene eseguito al momento della richiesta. Tipicamente viene usato per inserire blocchi di istruzioni condizionali o cicli. Viene eseguito nel momento in cui viene processata la richiesta. Oltre a poter accedere a dati e metodi dichiarati all'interno di tag di dichiarazione uno scriptlet consente di accedere ad alcuni oggetti impliciti ereditati dall'ambiente servlet.

```
<% if (Calendar.getInstance().get(Calendar.AM_PM) == Calendar.AM) { %>
Good Morning
<% } else { %>
Good Afternoon
<% } %>
...
<% if (request.getParameter ("utente").equals ("nuovo")) { %>
<B> Per favore eseguire la procedura di registrazione utente! </B>
<% } else { %>
<B> Bentornato !! </B>
<% } %>
```

Esempio 7.7 Scriptlet JSP

 Le **espressioni** sono denotate dalla sintassi <%= expression %>. Permettono l'inserimento di vere e proprie espressioni e variabili valutate a runtime. Quando l'espressione viene valutata, il risultato viene convertito in String e inserito nell'output HTML esattamente nel punto dove è definita. Il codice che segue inserisce la data corrente nel formato della macchina locale.

```
<%= (new java.util.Date( )).toLocaleString() %>
```

Esempio 7.8 Espressione JSP

7.2.3 Action

Le action forniscono informazioni per la fase di elaborazione della richiesta. L'interpretazione di un'azione può dipendere dai dettagli della specifica richiesta ricevuta dalla pagina JSP. Una action può essere **standard** (definita cioè nelle specifiche) o **custom** (fornita cioè tramite il meccanismo delle tag extensions o tag library).

La sintassi è quella di un elemento XML. Un elemento action ha perciò uno start tag che include il nome dell'elemento e può avere attributi, un corpo opzionale e un corrispondente end tag (es. `<mytag attr1="attribute value"...>body</mytag>`). Le azioni influenzano il comportamento della pagina JSP e della risposta inviata al client. Sono finalizzate ad un migliore incapsulamento del codice (per aderire allo spirito della programmazione ad oggetti Java), generalmente inteso come inclusione e utilizzo di Java Bean. Le principali azioni standard JSP sono le seguenti:

- **<jsp:useBean>** permette di utilizzare i metodi implementati all'interno di un bean.
- **<jsp:setProperty>** permette di impostare il valore di un parametro di un metodo di un bean.
- **<jsp:getProperty>** permette di acquisire il valore di un parametro di un bean.
- **<jsp:param>** permette di dichiarare ed inizializzare dei parametri all'interno della pagina. La sintassi è la seguente:

```
<jsp:params>
  <jsp:param name="nomeParametro" value="valore"/>
</jsp:params>
```

dove i parametri indicano:

- **name:** nome del parametro dichiarato.
- **value:** valore del parametro appena dichiarato.
- **<jsp:include>** permette di includere risorse aggiuntive di tipo sia statico sia dinamico. Il risultato è quindi quello di visualizzare la risorsa, oltre alle informazioni già inviate al client. La sintassi è la seguente:

```
<jsp:include page="URLRisorsa" flush="true | false" />
```

dove i parametri indicano:

- **page:** URL della risorsa da includere.
- **flush:** attributo booleano che indica se il buffer deve essere svuotato o no.

Questa azione può venire completata con la dichiarazione di eventuali parametri legati agli oggetti inclusi:

```
<jsp:include page="URLRisorsa" flush="true|false">
  {<jsp:param ... />}
</jsp:include>
```

- **<jsp:forward>** consente di eseguire una richiesta di una risorsa statica, un servlet o un'altra pagina JSP interrompendo il flusso in uscita. Il risultato è quindi la visualizzazione della sola risorsa specificata. La sintassi è la seguente:

```
<jsp:forward page="URLRisorsa"/>
```

dove **page** specifica l'URL della risorsa a cui eseguire la richiesta del servizio.

Ovviamente questa azione può anch'essa venire completata con la dichiarazione di parametri:

```
<jsp:forward page="URLRisorsa"/>
  {<jsp:param ... />}
</jsp:forward>
```

7.3 Commenti

Esistono due tipi di commento in una pagina JSP. Gli **Output Comment** sono inclusi all'interno dei tag `<!-- comment -->` e sono commenti spediti al client e visibili nel codice sorgente. Gli **Hidden Comment** sono commenti alla pagina JSP che documentano cosa fa la pagina, non sono spediti al browser e sono inclusi in `<%-- comment --%>`. Un modo alternativo per inserire un commento è usare il meccanismo fornito dal linguaggio di scripting (in Java `<% /** this is a comment ... */ %>`).

7.4 Gestione degli errori nelle pagine JSP

Una delle caratteristiche di Java è quella di poter gestire le **eccezioni**, cioè tutti quegli eventi che non dovrebbero accadere in una situazione normale e che non sono causati da errori da parte del programmatore. Dato che JSP deriva esplicitamente da Java, e ne conserva le caratteristiche di portabilità e robustezza, questo argomento non poteva essere trascurato.

7.4.1 Errori al momento della compilazione

Questo tipo di errore si verifica al momento della prima richiesta, quando il codice JSP viene tradotto in servlet. Generalmente sono causati da errori di compilazione ed il motore JSP, che effettua la traduzione, si arresta nel momento in cui trova l'errore, inviando al client richiedente una pagina di "Server Error" (codice di errore 500) con il dettaglio degli errori di compilazione.

7.4.2 Errori al momento della richiesta

Questi errori si verificano durante l'esecuzione della pagina e non in fase di compilazione. Si riferiscono all'esecuzione del contenuto della pagina o di qualche altro oggetto contenuto in essa. I programmatori Java sono abituati ad intercettare le eccezioni innescate da alcuni tipi di errori; nelle pagine JSP questo non è più necessario perché la gestione dell'errore in caso di eccezioni viene eseguita automaticamente dal servlet generato dalla pagina JSP. E' sufficiente creare un file .jsp che si occupi di gestire l'errore e che permetta in qualche modo all'utente di tornare senza troppi problemi all'esecuzione dell'applicazione JSP.

7.4.3 Creazione e uso di una pagina di errore

Una pagina di errore può essere vista come una normale pagina JSP in cui si specifica, tramite l'opportuno parametro della direttiva `page`, che si tratta del codice per gestire l'errore.

```
<HTML>
<BODY>
<%@ page isErrorPage = "true" %>
<CENTER>
Siamo spiacenti, si è verificato un errore
durante l'esecuzione:<BR><BR>
<%= exception.getMessage()%>
</CENTER>
</BODY>
</HTML>
```

Esempio 7.9 Una pagina di errore

A parte la direttiva `page`, il codice Java dell'esempio è composto da un'unica riga che utilizza l'oggetto **exception** (implicitamente contenuto in tutte le pagine di errore) richiamando il metodo **getMessage()**, il quale restituisce il messaggio di errore.

Perché in una pagina JSP venga utilizzata una determinata pagina di errore l'unica cosa da fare è inserire la direttiva `<% page errorPage = "PaginaErrore.jsp" %>`, che come è facile capire specifica quale pagina di errore deve essere richiamata in caso di errore in fase di esecuzione.

7.5 Oggetti e Scope

Una pagina JSP può creare, accedere e modificare **oggetti Java server-side** quando processa una richiesta. Tali oggetti possono essere visibili ad action ed elementi di scripting. Le specifiche indicano che alcuni oggetti sono creati implicitamente (**oggetti impliciti**), altri sono invece creati esplicitamente tramite action, altri ancora sono creati direttamente usando codice di scripting. Gli oggetti sono creati entro l'istanza di una pagina JSP che risponde ad una richiesta. Esistono diversi scope (ambiti):

- ❖ **page** – gli oggetti con questo ambito sono accessibili solo all'interno della pagina in cui sono stati creati; possono venire paragonati alle variabili locali di un qualsiasi linguaggio di programmazione. Tutti i riferimenti ad un tale oggetto sono rilasciati dopo che è stata chiusa la pagina. I riferimenti agli oggetti con scope page sono memorizzati nell'oggetto implicito **pageContext**.
- ❖ **request** – gli oggetti con questo ambito sono accessibili esclusivamente nelle pagine che elaborano la stessa richiesta di quella in cui è stato creato l'oggetto; quest'ultimo rimane nell'ambito anche se la richiesta viene inoltrata ad un'altra risorsa. I riferimenti all'oggetto sono rilasciati dopo che la richiesta è stata processata. I riferimenti agli oggetti con scope request sono memorizzati nell'oggetto implicito **request**.
- ❖ **session** – gli oggetti definiti in quest'ambito sono accessibili solo alle pagine che elaborano richieste all'interno della stessa sessione di quella in cui l'oggetto è stato creato, per poi venire rilasciati alla chiusura della sessione a cui si riferiscono; in pratica restano visibili in tutte le pagine aperte nella stessa istanza (finestra) del browser, fino alla sua chiusura. I riferimenti ad oggetti con scope session sono memorizzati nell'oggetto implicito **session**.
- ❖ **application** – gli oggetti definiti in quest'ambito sono accessibili alle pagine che elaborano richieste relative alla stessa applicazione; in pratica sono validi dalla prima richiesta di una pagina al server fino al suo shut-down. I riferimenti ad oggetti con scope application sono memorizzati nell'oggetto implicito **application**.

7.6 Oggetti Impliciti

JSP cerca di semplificare la costruzione di pagine fornendo alcuni oggetti accessibili in ogni pagina. Questi oggetti non necessitano di essere dichiarati né istanziati e sono forniti dal container stesso della pagina JSP. Tali oggetti impliciti sono dunque disponibili per essere usati entro scriptlet ed espressioni (non nelle dichiarazioni) attraverso le variabili che sono dichiarate implicitamente all'inizio della pagina. Per usufruire delle loro funzionalità è sufficiente usare la tipica sintassi nomeOggetto.nomeMetodo. La seguente tabella descrive gli oggetti impliciti disponibili.

Nome Variabile	Tipo	Semantica	Scope
request	Istanza di <code>javax.servlet.HttpServletRequest</code> .	Incapsula la richiesta che viene dal client e che verrà processata dal motore JSP. Viene passato dal container JSP come parametro del metodo <code>_jspService()</code> , dove viene modificato.	request
response	Istanza di <code>javax.servlet.HttpServletResponse</code>	Incapsula la risposta generata dal motore JSP per essere mandata al client in risposta alla richiesta effettuata. Viene generato dal motore JSP e passato come parametro al metodo <code>_jspService()</code> .	page
pageContext	Istanza di <code>javax.servlet.jsp.PageContext</code>	Incapsula il contesto della pagina.	page
session	Istanza di <code>javax.servlet.http.HttpSession</code>	Rappresenta la sessione creata per il client richiedente ed è valido solo per una richiesta HTTP. Gli oggetti sessione vengono creati automaticamente, l'unica eccezione avviene quando viene impostato come attributo di pagina "session=false": in questo caso cercando di referenziare una variabile di sessione si ottiene un errore di compilazione.	session

application	Istanza di <code>javax.servlet.ServletContext</code>	Rappresenta il contesto (ottenuto dall'oggetto <code>ServletConfig</code> come nella chiamata al metodo <code>getServletContext()</code>) nel quale la pagina JSP viene eseguita.	application
out	Istanza di <code>javax.servlet.jsp.JspWriter</code>	Rappresenta un oggetto che scrive nell'output stream. E' possibile impostare la dimensione del buffer o addirittura eliminarlo mediante la direttiva <code>page</code> e l'attributo <code>buffer</code> .	page
config	Istanza di <code>javax.servlet.ServletConfig</code>	Rappresenta la configurazione del servlet per la pagina JSP.	page
page	Istanza di <code>java.lang.Object</code>	E' l'istanza della implementation class della pagina JSP che sta processando la richiesta corrente. Quando il linguaggio di scripting è Java vi si può accedere usando la parola chiave "this".	page
exception	Istanza di <code>java.lang.Throwable</code>	E' disponibile solo nelle pagine di errore, quelle cioè nelle quali l'attributo <code>isErrorPage=true</code> .	page

Tabella 7.3 Oggetti impliciti disponibili nelle pagine JSP

7.6.1 L'oggetto request

L'oggetto implicito `request` permette di accedere alle informazioni di intestazione specifiche del protocollo HTTP. Al momento della richiesta, `request` incapsula le informazioni sulla richiesta del client e le rende disponibili attraverso alcuni suoi metodi.

L'uso più comune è quello di accedere ai parametri inviati (i dati provenienti da un form per esempio) con il metodo `getParameter("nomeParametro")`, che restituisce una stringa con il valore del parametro specificato. Altro metodo molto importante è `getCookies()`, che restituisce un array di cookies.

Gli altri metodi, i più importanti, sono: `getAttributeNames()` (restituisce una variabile di tipo `Enumeration` contenente i nomi di tutti gli attributi coinvolti nella richiesta), `getContentLength()` (restituisce un intero che corrisponde alla lunghezza in byte dei dati richiesti), `getContentType()` (restituisce il tipo MIME della richiesta), `getInputStream()` (restituisce un flusso di byte che corrisponde ai dati binari della richiesta; particolarmente utile per funzioni di upload di file da client a server), `getParameterNames()` (restituisce una variabile di tipo `Enumeration` contenente i nomi dei parametri della richiesta), `getParameterValues("nomeParametro")` (restituisce un array contenente tutti i valori del parametro specificato, nel caso ci siano più parametri con lo stesso nome), `getProtocol()` (corrisponde alla variabile d'ambiente `CGI_SERVER_PROTOCOL` e rappresenta il protocollo e la versione della richiesta), `getRemoteHost()` (corrisponde alla variabile d'ambiente `CGI_REMOTE_HOST`), `getServerName()` (corrisponde alla variabile `CGI_SERVER_NAME` e restituisce l'indirizzo IP del server), `getServerPort()` (corrisponde alla variabile `CGI_SERVER_PORT`, la porta alla quale in server è in ascolto), `getRemoteAddr()` (corrisponde alla variabile `CGI_REMOTE_ADDR` e restituisce in pratica l'indirizzo IP del visitatore), `getRemoteHost()` (corrisponde alla variabile `CGI_REMOTE_HOST`), `getRemoteUser()` (come per la variabile `REMOTE_USER` restituisce lo username della macchina richiedente), `getMethod()` (come per la variabile `REQUEST_METHOD` restituisce il metodo associato alla richiesta), `getPathInfo()` (restituisce informazioni extra sul path, corrisponde alla variabile `CGI_PATH_INFO`), `getQueryString()` (restituisce una stringa contenente l'intera query string, cioè tutti i caratteri dopo il punto di domanda, come per la variabile `CGI_QUERY_STRING`) e `getServletPath()` (restituisce il percorso relativo della pagina JSP).

7.6.2 L'oggetto response

Questo oggetto fornisce tutti gli strumenti per inviare i risultati dell'esecuzione della pagina JSP. I metodi principali sono `setBufferSize(intero)` (imposta la dimensione in byte del buffer per il corpo della risposta, escluse quindi le intestazioni), `setContentType("tipo")` (imposta il tipo MIME per la risposta al client) e `flushBuffer()` (forza l'invio dei dati contenuti nel buffer al client).

7.6.3 L'oggetto out

Questo oggetto ha principalmente funzionalità di stampa di contenuti. Con il metodo **print** è possibile stampare qualsiasi tipo di dato, così come con **println**, che a differenza del precedente termina la riga andando a capo. Si capisce comunque che l'andare o meno a capo nella stampa dei contenuti serve solo a migliorare la leggibilità del codice HTML.

7.6.4 L'oggetto pageContext

Questo oggetto rappresenta in pratica il contesto di esecuzione del servlet creato dalla pagina JSP. E' generalmente poco utilizzato dai programmatori di pagine JSP.

7.6.5 L'oggetto config

Questo oggetto permette di gestire tramite i suoi metodi lo startup del servlet associato alla pagina JSP: di accedere a parametri di inizializzazione e di ottenere riferimenti e informazioni sul contesto di esecuzione del servlet stesso.

7.6.6 L'oggetto exception

Questo oggetto è accessibile solo dalle pagine di errore (per le quali l'attributo `isErrorPage` è impostato a `true`). Contiene le informazioni relative all'eccezione sollevata in una pagina in cui il file è stato specificato come pagina di errore. Il metodo principale è **getMessage()** che restituisce una stringa con la descrizione dell'errore.

7.6.7 L'oggetto session

Una delle funzionalità più richieste per un'applicazione Web è mantenere le informazioni di un utente lungo tutto il tempo della sua visita al sito. Questo problema è risolto dall'oggetto implicito **session**, che gestisce appunto le informazioni a livello di sessione, relative ad un singolo utente, a partire dal suo ingresso fino alla sua uscita con la chiusura della finestra del browser. È possibile quindi creare applicazioni che riconoscono l'utente nelle varie pagine del sito, che tengono traccia delle sue scelte e dei suoi dati. È importante sapere che le sessioni vengono memorizzate sul server e non con dei cookie, che devono però essere abilitati per poter memorizzare il così detto **SessionID**. Questo consente di riconoscere il browser e quindi l'utente nelle fasi successive. I dati di sessione sono quindi riferiti e riservati ad uno specifico utente (per il quale viene creata una istanza dell'oggetto session) e non possono essere utilizzati da sessioni di altri utenti.

7.6.7.1 Memorizzare i dati nell'oggetto session

Per memorizzare i dati all'interno dell'oggetto implicito session è sufficiente utilizzare il metodo **setAttribute** specificando il nome dell'oggetto da memorizzare e una sua istanza. Per esempio, per memorizzare il nome dell'utente al suo ingresso alla pagina ed averlo a disposizione in seguito è sufficiente invocare il metodo `setAttribute("nomeUtente", nome)` a patto che nome sia un oggetto di tipo stringa contenente il nome dell'utente.

7.6.7.2 Leggere il contenuto di una variabile di sessione

La lettura di una variabile di sessione precedentemente memorizzata è possibile grazie al metodo **getAttribute**, il quale ha come unico ingresso il nome della variabile di sessione con cui è stato memorizzato il dato che interessa reperire. Ad esempio, `getAttribute("nomeUtente")` restituisce il nome dell'utente memorizzato come visto in precedenza; se non esiste la corrispondenza con il nome dato in ingresso, il metodo restituisce **null**.

7.6.7.3 Altri metodi dell'oggetto session

I due metodi `setAttribute` e `getAttribute` sono i metodi fondamentali e più utilizzati per la gestione della sessione; altri metodi dell'oggetto `session` potrebbero essere utili: `getAttributeNames()` (restituisce un oggetto di tipo enumerativo di stringhe contenente i nomi di tutti gli oggetti memorizzati nella sessione corrente), `getCreationTime()` (restituisce il tempo in millisecondi dalla mezzanotte del 1° gennaio 1970 relativo a quando è stata creata la sessione), `getId()` (restituisce una stringa contenente il SessionID che come detto permette di identificare univocamente una sessione), `getLastAccessedTime()` (restituisce il tempo in millisecondi dalla mezzanotte del 1° gennaio 1970 dall'ultima richiesta associata alla sessione corrente), `getMaxInactiveInterval()` (restituisce un valore intero che corrisponde all'intervallo massimo di tempo tra una richiesta dell'utente ad un'altra della stessa sessione), `removeAttribute(nome_attributo)` (rimuove l'oggetto dal nome specificato dalla sessione corrente).

7.6.8 L'oggetto application

L'oggetto `application` consente di accedere alle costanti dell'applicazione e di memorizzare oggetti a livello di applicazione, accessibili quindi da qualsiasi utente per un tempo che va dall'avvio del motore JSP alla sua chiusura, in pratica fino allo spegnimento del server. Gli oggetti memorizzati nell'oggetto `application` sono visibili da ogni utente e ogni pagina può modificarli. Per memorizzare i dati all'interno dell'oggetto `application` è sufficiente utilizzare il metodo `setAttribute` specificando il nome dell'oggetto da memorizzare e una sua istanza. La lettura di un oggetto `application` precedentemente memorizzato è possibile grazie al metodo `getAttribute` che ha come unico parametro il nome dell'oggetto `application` con cui avevamo memorizzato il dato che ci interessa reperire. Se non vengono trovate corrispondenze con il nome restituisce il valore `null`.

Anche l'oggetto `application`, come `session`, possiede il metodo `getAttributeNames()`, che restituisce un oggetto di tipo enumerativo di stringhe contenente i nomi di tutti gli oggetti memorizzati nella applicazione in esecuzione. Per rimuovere un oggetto si utilizza `removeAttribute("nomeoggetto")`.

7.7 Utilizzo dei JavaBeans

Le pagine JSP sono estensioni dei servlet Java; di conseguenza consentono di ottenere tutti i risultati di questi ultimi. Lo svantaggio dei servlet è rappresentato, come per tutti gli script CGI, dalla difficoltà di manutenzione del codice HTML delle applicazioni eccessivamente complesse, in quanto il più delle volte contengono una quantità enorme di codice. Inoltre, per quanto riguarda JSP, includere troppo codice all'interno di una pagina comporta comunque una ridotta leggibilità dello stesso ed una non netta separazione fra la parte di presentazione dei contenuti e la parte di creazione degli stessi. Un altro svantaggio si ha quando un blocco di codice serve in più di una pagina: inserendo il codice direttamente nella pagina occorrerebbe ripeterlo ogni volta.

La soluzione a questi problemi è rappresentata dai **JavaBeans** (o bean), componenti software contenenti una classe Java, che possono venire inclusi in una pagina JSP, permettendo quindi un ottimo incapsulamento del codice, peraltro riutilizzabile. La sezione di codice puro, sostituito da richiami ai metodi delle classi incluse, sarà pressoché invisibile al programmatore. I bean sono costituiti esclusivamente da codice Java, e possono essere realizzati con un semplice editor di testo, salvati con estensione `.java` e compilati, ad esempio con il compilatore `javac` incluso nel JDK: questo genererà un file di estensione `.class`. È il file `.class` che sarà incluso nella pagina JSP, come si vedrà in seguito. Si ricorda che in Java i file devono avere lo stesso nome della classe, lettere maiuscole e minuscole comprese (case-sensitive). Utilizzando i bean si riesce a rimuovere gran parte del codice da una pagina JSP. Il risultato è una pagina più semplice sia da leggere sia da mantenere. La tecnologia JavaBeans è quindi preziosa per costruire componenti portabili e riusabili che possono essere usati con la tecnologia JSP. Sono diversi i modi di usare i componenti JavaBeans all'interno di una applicazione. Innanzi tutto possono essere usati come oggetti data-centric, creati specificamente per manipolare e ritornare dati; in questo caso possono essere usati da diverse view di un'applicazione e da molti client contemporaneamente. Unito ad un Front Component, un bean può essere usato come controller (es. bean usato per processare tutte le richieste ricevute dal FC e passarle alla pagina appropriata). Infine, componenti JavaBeans page-specific forniscono la logica per processare i dati e generare un risultato per una particolare pagina. Lo svantaggio, in questo caso, è che tali componenti sono difficili da riusare.

7.7.1 Il modello JavaBeans e le sue caratteristiche fondamentali

Un JavaBeans è un componente software riutilizzabile (classe Java) che può essere manipolato visualmente in un tool di sviluppo software. Il modello è gestito dal package **java.beans** e le sue parti fondamentali sono:

- **Metodi** ossia funzioni che permettono di accedere all'istanza del bean per ottenere e/o manipolare informazioni.
- **Proprietà** di lettura e/o scrittura che rappresentano gli attributi dell'oggetto.
- **Eventi** che un bean può segnalare o ricevere dal mondo esterno (es. il click su un bottone).

Il modello possiede inoltre:

- Un supporto per l'**introspezione**, la caratteristica che permette ai tool di sviluppo di ottenere informazioni su metodi, proprietà ed eventi di un particolare bean. L'introspezione viene gestita dalla classe **java.beans.Introspector**.
- Un supporto per la **personalizzazione**, in modo che il programmatore possa modificare lo standard di default stabilito per il bean e fargli assumere la configurazione desiderata. La personalizzazione di un bean riguarda due aspetti fondamentali:
 - a. *Modifica dello standard definito per metodi, proprietà ed eventi.* Questo aspetto viene realizzato costruendo una classe di nome `<MioBean>BeanInfo` che implementa la interfaccia **java.beans.BeanInfo** o estende la classe **java.beans.SimpleBeanInfo**, dove `<MioBean>` è il nome che è stato dato al bean.
 - b. *Aiuto nella creazione del bean.* Tale aiuto consiste nel progettare una serie di wizard (maschere) che guidano l'utente nell'impostazione di metodi, proprietà ed eventi, generando alla fine il codice relativo all'istanza del bean desiderata. Questa utilità viene ottenuta implementando l'interfaccia **java.beans.Customizer**.
- Un supporto per la **persistenza**, caratteristica che consiste nella possibilità di salvare e ripristinare informazioni di un'istanza di un oggetto in un file su disco. Quando invece di salvare le informazioni su disco le si spedisce attraverso la rete, si parla di **serializzazione**, molto usata dal protocollo RMI per ricevere i parametri di un metodo remoto invocato dal client e restituirgli indietro il risultato sotto forma di un'istanza di un altro oggetto. Per dichiarare una classe serializzata basta che questa estenda l'interfaccia **java.io.Serializable**. La scrittura e la lettura dell'istanza dell'oggetto viene realizzata utilizzando rispettivamente il metodo **writeObject** della classe **java.io.ObjectOutput** e il metodo **readObject** della classe **java.io.ObjectInput**.

Alla fine sia per minimizzare lo spazio occupato da tutte le classi del bean, e di eventuali altri file che il bean potrebbe gestire (immagini, suoni, ecc.), sia per velocizzare il tempo di trasferimento di tali file attraverso la rete, il tutto viene compresso in un file con estensione `.jar` (Java Archive) ottenuto dal comando `jar` presente nella directory **bin** del pacchetto JDK.

7.7.2 Proprietà dei bean

Un bean usa le proprietà per descrivere i dati interni che influenzano quello che fa e quello che mostra. La convenzione vuole che le variabili all'interno di un bean siano inaccessibili dall'esterno in maniera diretta (sono quindi dichiarate di tipo **private** o **protected**) ed accessibili mediante appositi **metodi public**, in completo accordo anche con la programmazione orientata agli oggetti, dove i dati interni sono nascosti all'utente e vengono esposti solo attraverso metodi accessori. I metodi usano la seguente sintassi convenzionale:

- ◆ `public void setNomeProprietà(tipoProprietà valore) // imposta il valore della proprietà`
- ◆ `public tipoProprietà getNomeProprietà()`
- ◆ `public boolean isNomeProprietà() // usato se la proprietà è di tipo booleano`
- ◆ `public tipoProprietà getNomeProprietà(int index)`
- ◆ `public void setNomeProprietà(int index, tipoProprietà valore)`
- ◆ `public tipoProprietà[] getNomeProprietà()`
- ◆ `public void setNomeProprietà(tipoProprietà[] valore)`

7.7.3 Aggiunta di un bean in una pagina JSP

Esistono tre azioni standard che facilitano l'integrazione dei JavaBeans nelle pagine JSP, già introdotte nella sezione relativa alle azioni.

- **<jsp:useBean>** permette di associare un'istanza di un JavaBeans (associata ad un determinato ID) ad una variabile script dichiarata con lo stesso ID. In pratica offre la possibilità di associare la classe contenuta nel JavaBeans ad un oggetto visibile all'interno della pagina, in modo da poter richiamare i suoi metodi senza dover ogni volta far riferimento al file di origine. La sintassi è la seguente:

```
<jsp:useBean id="name" scope="page|request|session|application" class="classe" />
```

dove i parametri indicano:

- **id**: identità dell'istanza dell'oggetto nell'ambito specificato.
- **scope**: ambito dell'oggetto.
- **class**: nome della classe che definisce l'implementazione dell'oggetto bean; contiene il nome del bean che deve coincidere con il nome del file .class (senza estensione)

Per esempio la action `<jsp:useBean id="nomeBean" scope="session" class="prova" />` crea un'istanza della classe prova con ambito session richiamabile attraverso l'id nomeBean; da questo momento sarà possibile accedere a metodi e variabili (pubbliche) del bean per mezzo della classica sintassi nomeBean.nomeMetodo e nomeBean.nomeVariabile, rispettivamente per metodi e variabili.

- **<jsp:setProperty>** permette di impostare il valore di una delle proprietà di un bean. Le sintassi possibili sono le seguenti:

```
<jsp:setProperty name="beanName" property="propertyName" param="parameterName" />
```

```
<jsp:setProperty name="beanName" property="propertyName" value="propertyValue" />
```

dove i parametri indicano:

- **name**: nome dell'istanza di bean definita in un'azione.
- **property**: proprietà di cui impostare il valore.
- **param**: nome del parametro di richiesta il cui valore si vuole impostare.
- **value**: valore assegnato alla proprietà specificata.

Per esempio `<jsp:setProperty name="nome_bean" property="prop" param="nome_parametro" />` permette di assegnare il valore del parametro nome_parametro alla proprietà prop del bean di nome nome_bean.

- **<jsp:getProperty>** prende il valore di una proprietà di una data istanza di bean e lo inserisce nell'oggetto implicito out (in pratica lo stampa a video). La sintassi è la seguente:

```
<jsp:getProperty name="name" property="propertyName" />
```

dove i parametri indicano:

- **name**: nome dell'istanza di bean da cui proviene la proprietà definita da un'azione.
- **property**: proprietà del bean di cui si vuole ottenere il valore.

7.7.4 Un esempio di bean

Il bean proposto di seguito è realizzato per contenere alcune informazioni di un utente durante la sua permanenza in un sito.

```
public class InfoUtente {
    private String nome = null;
    private String email = null;
    private int pagineViste;
    public InfoUtente( ) {
        pagineViste=0;
    }
}
```

```

public aggiornaPV( ){
    pagineViste++; // incrementa il numero di pagineViste
}
public int getPagineViste( ){
    return pagineViste; // ritorna il numero di pagineViste
}
public void setNome(String value) {
    nome = value; // imposta il nome dell'utente
}
public String getNome( ) {
    return nome; // restituisce il nome dell'utente
}
public void setEmail(String value) {
    email = value; // imposta l'email dell'utente
}
public String getEmail( ) {
    return email; // restituisce l'email dell'utente
}
public String riassunto( ){
    String riassunto = null;
    riassunto = "Il nome dell'utente è "+nome+";";
    riassunto+= "il suo indirizzo e-mail è: "+email;
    riassunto+=" e ha visitato "+pagineViste+" del sito";
    return riassunto;
}
}
} //InfoUtente

```

Esempio 7.10 Il bean InfoUtente.java

Com'è facile capire, il bean InfoUtente contiene il nome dell'utente ed i metodi per modificarlo e restituirlo, il suo indirizzo e-mail con i relativi metodi, il numero di pagine visitate dall'utente e un metodo che restituisce un riassunto schematico dei dati dell'utente. Ecco come utilizzarli.

```

<html>
<head><title>Utilizzo del Bean</title></head>
<body>
<jsp:useBean id="utente" scope="session" class="InfoUtente"/>

```

Esempio 7.11 Creazione di un'istanza del bean in JSP

Con il codice appena mostrato viene creata un'istanza del bean InfoUtente con ambito session, necessario per questo tipo di funzione.

```

<jsp:setProperty name="utente" property="nome" value="FabioRomba"/>
- oppure -
<%
utente.setNome("FabioRomba");
utente.setEmail("fabioromba@katamail.com");
%>

```

Esempio 7.11 Impostazione delle proprietà del bean

Le proprietà del bean possono essere impostate con l'azione setProperty o agendo direttamente con i metodi creati appositamente.

Lo stesso vale per la lettura delle proprietà del bean, la quale può essere fatta con l'azione getProperty o richiamando i metodi messi a disposizione appositamente dal bean. Il codice che segue mostra come, usando i due meccanismi appena citati, è possibile leggere l'attributo (proprietà) nome del bean InfoUtente.

```
<jsp:getProperty name="utente" property="nome"/>
```

- oppure -

```
<% out.println(utente.getNome( ));  
out.println(utente.riassunto( )); %>
```

Esempio 7.12 Lettura delle proprietà del bean

Per incrementare il numero di pagine visitate è sufficiente richiamare il metodo `aggiornaPV()` e per ottenerne il valore `getPagineViste()`.

```
<% utente.aggiornaPV( );  
out.println(utente.getPagineViste( )); %>
```

Esempio 7.13 Chiamate a metodi definiti nel bean

7.8 Breve accenno alle Tag Library

I **custom tag** sono il meccanismo fornito dalla tecnologia JSP per definire funzionalità customizzate, dichiarative e modulari usate dalle pagine JSP. Sono conosciuti come **tag library** ed importati in una pagina JSP usando la direttiva `taglib`. Una volta importato, un custom tag può essere usato nella pagina servendosi del prefisso definito dalla direttiva.

I custom tag forniscono la stessa funzionalità dei bean. Comunque, al contrario dei componenti JavaBeans, i quali devono essere prima dichiarati e poi acceduti usando metodi `get` e `set`, i custom tag ottengono le informazioni di inizializzazione dai parametri definiti quando il tag è creato. Essi hanno accesso al Web container e a tutti gli oggetti disponibili per le pagine JSP. Possono modificare la risposta generata. Inoltre sono portabili e riusabili, essendo scritti in Java.

Capitolo 8 L'Enterprise JavaBeans Tier

In un'applicazione J2EE multi-tier, l'EJB Tier ospita la business logic specifica delle applicazioni e fornisce specifici servizi system-level (gestione delle transazioni, controllo della concorrenza, e sicurezza). La tecnologia Enterprise JavaBeans fornisce un modello a componenti distribuiti che permette agli sviluppatori di focalizzarsi esclusivamente sulla soluzione dei business problem lasciando alla piattaforma J2EE la gestione dei complessi problemi system-level.

Questa separazione di ambiti permette il rapido sviluppo di applicazioni scalabili, facilmente accessibili ed altamente sicure. Nel modello di programmazione di J2EE, i componenti EJB sono un link fondamentale tra i componenti di presentation residenti nel Web Tier ed i dati e sistemi business-critical mantenuti nell'EIS Tier. La tecnologia EJB ha svariati vantaggi:

- Semplificazione del processo di sviluppo
- Riutilizzabilità del codice e modularità
- Robustezza
- Gestione automatica di:
 - transazioni (Commit, Rollback e Recovery)
 - scalabilità (aumentando l'HW, le prestazioni aumentano in modo lineare)
 - sicurezza
- Alte prestazioni
 - bilanciamento dinamico dei carichi di lavoro
 - caching delle connessioni al database

8.1 La Business Logic

La **business logic** è, in senso ampio, il set di linee guida per realizzare una business function. Rifacendosi all'approccio object-oriented permette di decomporre una business function in un set di componenti od elementi chiamati **business object**. Come gli altri oggetti, questi ultimi avranno caratteristiche (stato e dati) e comportamento. Per esempio, un employee object avrà dati come nome, indirizzo, telefono, codice fiscale, e così via. Avrà metodi per assegnare l'impiegato ad un nuovo dipartimento o per cambiare il suo salario di una certa cifra (percentuale). Dobbiamo poter rappresentare in che modo questi oggetti funzionano o interagiscono per fornire la funzionalità desiderata. Le regole business-specific che ci aiutano ad identificare la struttura ed il behavior (comportamento) dei business object e le loro interazioni con altri oggetti sono conosciute come business logic. Quelli di seguito elencati sono requisiti di esempio che devono essere soddisfatti da un account object: è a partire da loro che deve essere definita la struttura ed il behavior del business object associato.

1. Ogni cliente deve avere un unico account
2. Ogni account dovrebbe contenere informazioni di un cliente quali nome, indirizzo ed e-mail
3. I clienti devono poter creare nuovi account
4. I clienti devono poter aggiornare le informazioni relative ai loro account
5. I clienti devono poter ricevere informazioni sui loro account
6. I clienti possono aggiornare o ricevere informazioni solo sui loro account
7. Le informazioni relative a ciascun account devono essere mantenute in una memoria persistente
8. Molteplici clienti possono accedere alle informazioni sui loro account contemporaneamente
9. Più clienti non possono aggiornare lo stesso account concorrentemente

I primi due requisiti specificano attributi strutturali dell'account object. Seguendo queste regole, l'account object dovrebbe avere un campo per poter essere identificato univocamente e altri campi per memorizzare indirizzo, telefono, ecc... Il behavior dell'oggetto è descritto dai requisiti 3,4 e 5. Ad esempio si devono prevedere metodi per creare un nuovo account, aggiornare le informazioni, ecc... Gli ultimi quattro specificano condizioni generali di cui tenere conto quando si realizza il behavior dell'account object. Per esempio, quando un cliente aggiorna un account deve essere autorizzato ad accedere al particolare account.

8.1.1 Requisiti comuni dei Business Object

1. Mantenere lo stato

Un business object spesso deve mantenere lo stato rappresentato dalle sue variabili di istanza tra le chiamate a metodi. Lo stato di un business object può essere di due tipi: **conversazionale** o **persistente**. Si consideri un shopping cart object (carrello della spesa virtuale). Lo stato di tale oggetto rappresenta gli articoli e le quantità di ciascun articolo acquistate dal cliente. Il carrello è inizialmente vuoto e raggiunge uno stato significativo quando un utente vi aggiunge un articolo. Quando un utente aggiunge un altro articolo al carrello, il carrello conterrà entrambi gli articoli. Allo stesso modo, quando un utente cancella un articolo dal carrello, il carrello rifletterà il cambiamento del suo stato. Quando un utente esce dall'applicazione, l'oggetto deve essere reinizializzato. Quando l'oggetto cambia, mantiene o perde il suo stato come risultato di ripetute interazioni con lo stesso client si dice che mantiene uno stato conversazionale. Per capire lo stato persistente, si consideri un'account object. Quando un utente crea un account, le informazioni relative devono essere memorizzate permanentemente, così che quando l'utente esce dall'applicazione e vi rientra, le informazioni sull'account possono essere nuovamente presentate all'utente. Lo stato di un account object deve essere mantenuto in una memoria permanente come un database.

2. Elaborare dati condivisi

Spesso i business object operano su dati condivisi. In tal caso, devono essere previsti controllo di concorrenza e appropriati livelli di isolation di tali dati. Un esempio di tale scenario potrebbe essere il caso in cui utenti multipli aggiornano le stesse informazioni di account. Se due utenti tentano di aggiornare lo stesso account contemporaneamente, il business object deve fornire un meccanismo per mantenere i dati in uno stato consistente.

3. Supportare le transazioni

Una transazione può essere descritta come un insieme di processi (task) che devono essere completati come un'unità. Se uno dei processi fallisce, si deve eseguire il rollback di tutti i processi che compongono la transazione. Se tutti i processi hanno successo, si deve eseguire il commit della transazione. Le proprietà delle transazioni sono denotate dall'acronimo **ACID**.

- ✘ **Atomicity** richiede che tutte le operazioni di una transazione siano portate a termine con successo perché la transazione sia considerata completa. Se almeno una di tali operazioni non può essere realizzata, allora nessuna deve essere portata a termine.
- ✘ **Consistency** si riferisce alla consistenza dei dati. Una transazione deve portare i dati da uno stato consistente all'altro, proteggendo la semantica e l'integrità fisica dei dati.
- ✘ **Isolation** richiede che ciascuna transazione sembri la sola che sta in quel preciso istante manipolando i dati. Le altre transazioni possono girare in maniera concorrente.
- ✘ **Durability** significa che gli aggiornamenti fatti da una transazione di cui si è fatto il commit persistono nel database indifferentemente dai fallimenti occorsi dopo l'operazione di commit ed assicura che i database possono essere recuperati dopo un failure del sistema.

I business object spesso partecipano alle transazioni. Per esempio, una disposizione per un ordine ha bisogno di essere transazionale visto l'insieme di processi richiesti per completare un ordine: decrementare la quantità degli articoli comprati dall'inventario, memorizzare i dettagli dell'ordine, e spedire una conferma dell'ordine all'utente. Affinché la transazione possa essere completata, tutti questi processi devono aver successo. Se uno qualsiasi fallisce, il lavoro fatto dagli altri processi deve essere annullato (undo).

In molti casi, le transazioni possono riguardare più di una fonte dati remota. Tali transazioni – chiamate transazioni distribuite – richiedono speciali protocolli per assicurare l'integrità dei dati (es. tabella dell'inventario e tabella degli ordini residenti in differenti database).

4. Servire un largo numero di client

Un business object dovrebbe poter fornire il suo servizio ad un elevato numero di client contemporaneamente. Ciò si traduce in un requisito per gli algoritmi di gestione dell'istanza che dia a ciascun client l'impressione che la propria richiesta sia servita da un business object dedicato (multithreading).

5. Fornire accesso remoto ai dati

Un client dovrebbe poter accedere remotamente ai servizi offerti da un business object. Questo significa che il business object dovrebbe avere un qualche tipo di infrastruttura per fornire il servizio ai client attraverso la rete. Ciò in breve implica che un business object dovrebbe far parte di un ambiente distribuito che si occupi di problemi fondamentali presenti nei sistemi distribuiti come trasparenza della location e del failure.

6. Controllare l'accesso

I servizi offerti dai business object spesso richiedono un meccanismo di autenticazione ed autorizzazione del client per l'accesso protetto ai servizi. Per esempio, un account object deve validare l'autenticità del client prima di permettergli di aggiornare le informazioni riguardanti il suo account. In molti scenari sono necessari diversi livelli di controllo degli accessi; ad esempio gli impiegati possono visualizzare solo il loro salary object, mentre un amministratore delle paghe può sia visualizzare sia modificare tutti i salary object.

7. Essere riusabile

Un requisito comune dei business object è che devono essere riusabili da differenti componenti della stessa applicazione e/o da differenti applicazioni. Per esempio, un'applicazione usata dal dipartimento paghe per tener traccia dei salari degli impiegati può avere due business object: employee object e salary object. Un salary object può usare i servizi forniti da un employee object per ottenere l'inquadramento di un impiegato. Un'applicazione che tiene traccia delle ferie e delle indennità di ferie degli impiegati può voler usare lo stesso employee object per ottenere il nome dell'impiegato attraverso il suo codice. Per poter essere usati da componenti inter- e intra-application, i business object devono essere sviluppati in un modo standard e devono girare in un ambiente che supporti questo standard. Se lo standard viene largamente adottato dalla comunità di produttori, un'applicazione può essere assemblata usando componenti forniti da differenti vendor.

8.2 Enterprise Bean come J2EE Business Object

Per semplificare lo sviluppo, le applicazioni hanno bisogno di una infrastruttura server-side standard che possa fornire servizi come supporto alle transazioni, sicurezza ed accesso remoto ai dati. La tecnologia fornita da Sun per implementare i business object è quella comunemente indicata come **Enterprise JavaBeans**. L'EJB Tier della piattaforma J2EE fornisce un modello a componenti server-side distribuiti standard che semplifica enormemente il processo di scrittura della business logic. Per usare i servizi forniti dalla piattaforma J2EE, i business object sono implementati come componenti EJB, o **enterprise bean**. Tali componenti si occupano di:

- ✗ ricevere dati da un client, processare tali dati (se necessario), inviare i dati all'EIS Tier per la loro memorizzazione su database;
- ✗ (viceversa) acquisire dati da un database appartenente allo strato EIS, processare tali dati (se necessario), inviare tali dati al programma client che n'abbia fatto richiesta.

Esistono due tipi principali di enterprise bean: session bean e entity bean. I **session bean** sono oggetti non persistenti che rappresentano una risorsa privata rispetto al client che li ha creati. Per questa ragione, i session bean, dalla prospettiva dell'utente, appaiono anonimi. Al contrario, tutti gli **entity bean** sono oggetti persistenti, rappresentano in modo univoco un dato esistente all'interno dello strato EIS ed hanno quindi una loro precisa identità, rappresentata da una chiave primaria. In aggiunta ai componenti, l'architettura EJB definisce altre entità: server, container e client. Gli enterprise bean vivono all'interno di **EJB container**, che forniscono gestione del ciclo di vita e una varietà di altri servizi quali gestione della concorrenza e scalabilità. Un EJB container è a sua volta parte di un **EJB server**, che fornisce servizi di naming e directory, di e-mail ed altro ancora. Quando un client invoca un'operazione su un enterprise bean la chiamata è intercettata dal suo container. Intercedendo tra client e componente a livello di chiamata del metodo, il container può gestire servizi quali la propagazione della chiamata ad altre componenti (load balancing) o ad altri container (scalabilità) su altri server sparsi per la rete su differenti macchine. Questo meccanismo semplifica lo sviluppo sia dei componenti sia dei client.

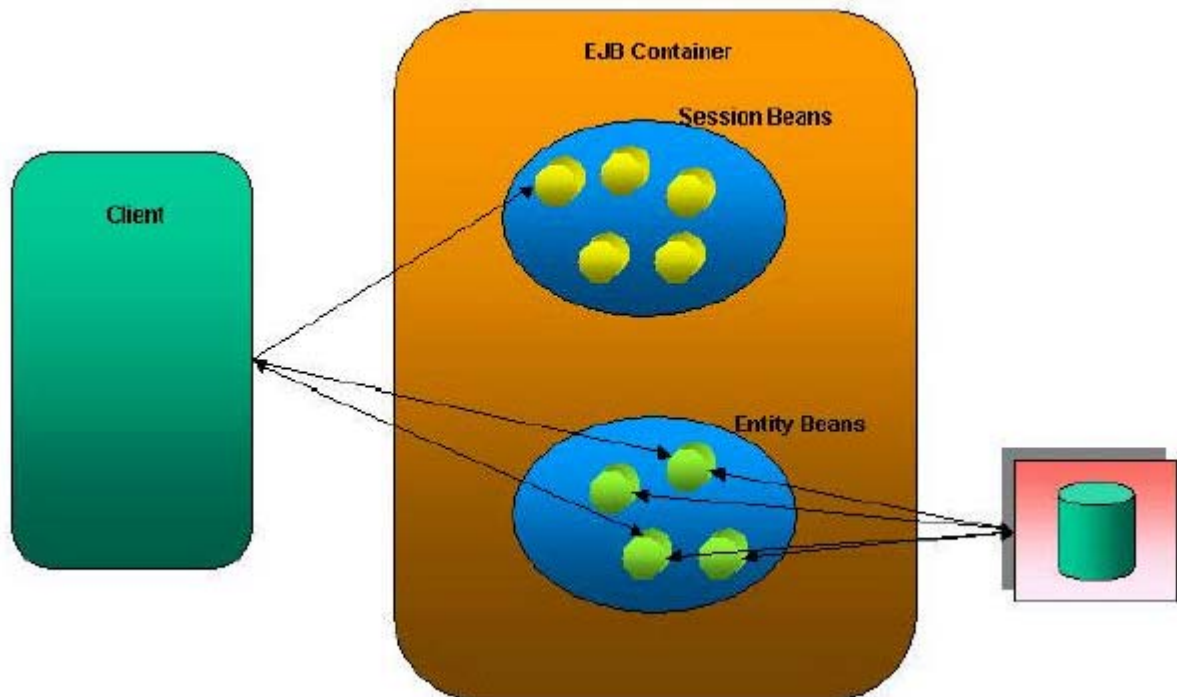


Figura 8.1 L'EJB Tier

Nell'implementazione di un'architettura enterprise, oltre a decidere che tipo di enterprise bean utilizzare, si devono effettuare altre scelte strategiche nella definizione del modello a componenti: che tipo di oggetto debba rappresentare un enterprise bean e che ruolo tale oggetto deve avere all'interno di un gruppo di componenti che collaborano tra loro. Poiché gli enterprise bean sono oggetti che richiedono abbondanti risorse di sistema e di banda di rete, non sempre è soluzione ottima modellare tutti i business object come EJBs. In generale una soluzione consigliabile è quella di adottare tale modello solo per quelle componenti che richiedano un accesso diretto da parte di un client.

8.2.1 L'architettura EJB

L'EJB server (o Application Server) è il motore che permette l'utilizzo di componenti dai client remoti. Fornisce ai container i seguenti servizi:

- Gestione delle risorse di sistema
- Mantenimento dello stato
- Gestione e attivazione dei processi e thread
- Sicurezza

Un EJB server gestisce le politiche di ottimizzazione delle prestazioni come pooling e caching delle connessioni a database, clustering, fault tolerance, caching dei bean.

L'EJB container ospita ed esegue gli EJB. L'enterprise bean è il componente vero e proprio: è una classe Java specializzata in cui risiede la business logic dell'applicazione. Tale componente deve essere installato sull'Application Server e utilizza i servizi offerti dall'ambiente di esecuzione: transazioni, sicurezza, persistenza.

8.2.2 Enterprise Bean e EJB Container

L'architettura EJB fornisce agli enterprise bean e agli EJB container di diverse caratteristiche, le quali esaltano la portabilità ed il riutilizzo.

- ✘ Le istanze di un enterprise bean sono create e gestite runtime da un container.
- ✘ Il comportamento di un enterprise bean non è interamente contenuto nell'implementazione. Informazioni di servizio (service information), incluse transazioni e sicurezza, sono separate

dall'implementazione. Questo permette di customizzare le informazioni di servizio al momento dell'assemblaggio e del deployment dell'applicazione. Il behavior dell'enterprise bean è customizzato a deployment time tramite il suo deployment descriptor. Ciò permette di includere un enterprise bean in un'applicazione senza richiedere cambiamenti al codice o ricompilazione.

- ✘ Il Bean Provider definisce una **client view** dell'enterprise bean, che non è influenzata dal container e dal server nei quali il bean è deployato. Questo assicura che sia i bean sia i loro client possono essere deployati in molteplici ambienti senza la necessità di cambiamenti o ricompilazione. La client view è fornita attraverso due interfacce (**home interface** e **remote interface**), implementate dalle classi costruite dal container quando il bean è deployato (**enterprise bean class**). Implementando queste interfacce il container può intercedere nelle operazioni del client su un bean e offrire una vista semplificata del componente.

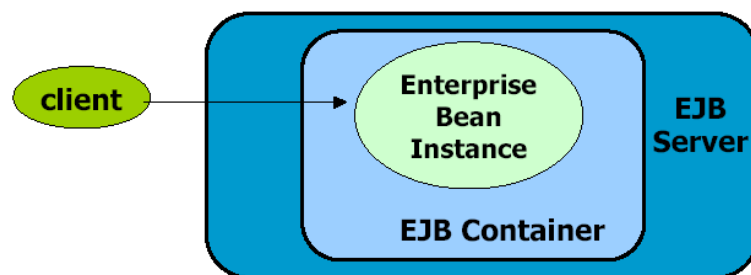


Figura 8.2 L'istanza di un enterprise bean è creata e gestita runtime da un container

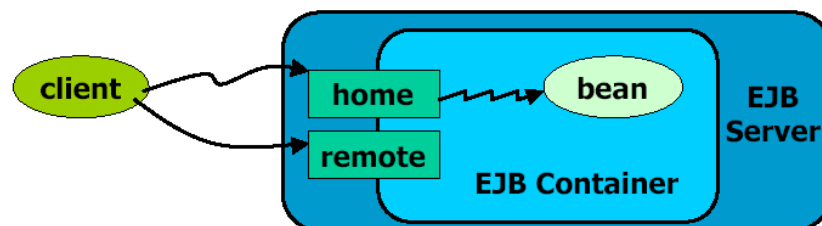


Figura 8.3 La client view di un enterprise bean è fornita attraverso le interfacce home e remote

8.2.2.1 La Home Interface

La **home interface** fornisce metodi per creare e rimuovere enterprise bean. Questa interfaccia deve estendere **javax.EJB.EJBHome** e permette ad un client di:

- ✘ Creare nuove istanze di un enterprise bean
- ✘ Rimuovere un'istanza di un enterprise bean
- ✘ Ottenere meta-data per l'enterprise bean attraverso l'interfaccia **javax.ejb.EJBMetaData**, fornita per permettere ai tool per l'assemblaggio di ottenere informazioni sull'enterprise bean al momento del deployment.
- ✘ Ottenere un handle per la home interface, che fornisce i meccanismi necessari agli enterprise bean persistenti. Inoltre, l'home interface di un entity bean fornisce metodi per trovare istanze esistenti di un entity bean entro la home. Un client che conosce la primary key di un entity bean può ottenere un riferimento ad esso invocando un particolare metodo: **findByPrimaryKey**.

Dunque **javax.ejb.EJBHome** definisce i metodi che permettono al client di creare, trovare e rimuovere gli oggetti EJB. Ciascun enterprise bean ha un'interfaccia home che è un'estensione di **EJBHome**.

L'interfaccia home, in parole povere, è quella che ha il compito di gestire l'oggetto EJB che contiene i metodi remoti attivati dal client.

8.2.2.2 La Remote Interface

La **remote interface** definisce la client view di un enterprise bean, cioè l'insieme di metodi disponibili per il client. Questa interfaccia deve estendere **javax.ejb.EJBObject**. Un EJBObject supporta i metodi business dell'oggetto. L'EJBObject delega la chiamata di un metodo business all'istanza dell'enterprise bean. L'interfaccia javax.ejb.EJBObject definisce i metodi che permettono ai client di realizzare le seguenti operazioni con riferimento all'istanza di un enterprise bean:

- ✗ Ottenere la home interface
- ✗ Rimuovere l'istanza dell'enterprise bean
- ✗ Ottenere un handle per l'istanza dell'enterprise bean
- ✗ Ottenere la primary key dell'istanza di un entity bean

Le estensioni di javax.ejb.EJBObject consistono quindi di interfacce che contengono tutti i metodi remoti che saranno richiamati dal client. Un client non accede mai alle istanze di un enterprise bean direttamente ma usa sempre un'interfaccia remota per accedere alle istanze. La classe che implementa l'interfaccia remota EJBObject è fornita dal contenitore.

8.2.2.3 La Enterprise Bean Class

La **enterprise bean class** fornisce l'implementazione effettiva dei metodi business del bean. E' chiamata dal container quando il client chiama i corrispondenti metodi elencati nella remote interface. Questa classe deve implementare l'interfaccia **javax.ejb.EntityBean** o l'interfaccia **javax.ejb.SessionBean**. In aggiunta ai metodi business, la remote interface e la enterprise bean class condividono la responsabilità per due categorie specializzate di metodi: **metodi create**, che permettono di customizzare il bean nel momento in cui è creato, e **metodi finder**, che forniscono modi per localizzare un bean. Per ogni metodo create elencato nella home interface, la enterprise bean class implementa il corrispondente metodo **ejbCreate**. Per ogni metodo finder elencato nella home interface, la enterprise bean class implementa il corrispondente metodo **ejbFindBy...**

In aggiunta, l'enterprise bean class deve fornire le implementazioni dei metodi elencati nella interfaccia che estende. Uno sviluppatore può scegliere di fornire implementazioni vuote di qualsiasi metodo non richiesto per lo scopo specifico del bean.

La Figura 8.4 illustra l'implementazione della client view di un enterprise bean.

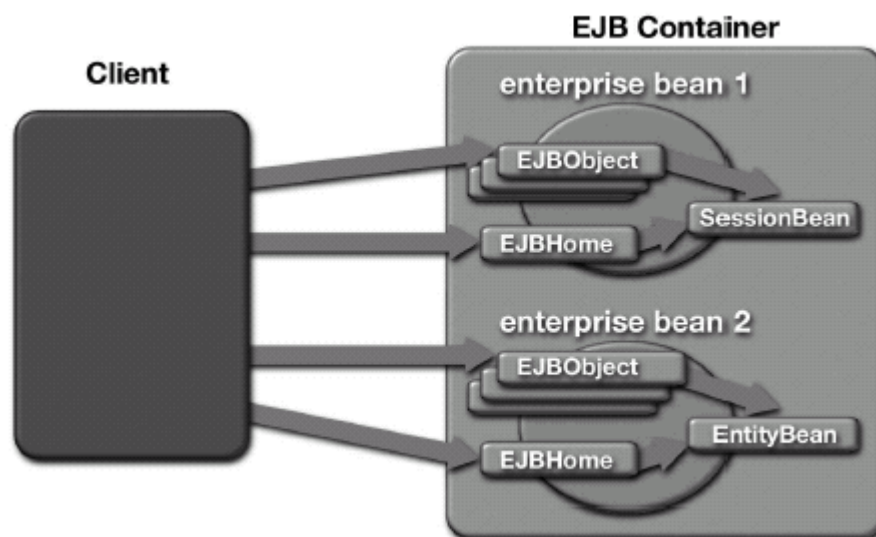


Figure 8.4 Implementazione della client view di un enterprise bean

8.2.2.4 Introduzione a Entity Bean e Session Bean

javax.ejb.EnterpriseBean è il fulcro del modello EJB. Fornisce un servizio al client di tipo transazionale o di accesso ai dati. L'architettura EJB, come già accennato, definisce due tipi di enterprise bean, i quali sono estensioni di javax.ejb.EnterpriseBean.

javax.ejb.EntityBean è un oggetto di tipo entità con le seguenti caratteristiche:

1. Rappresenta i dati in un database.
2. E' transazionale.
3. Permette accessi condivisi da più client.
4. Può avere una lunga vita (vive tanto quanto i dati nel database).
5. Sopravvive ai crash di un server EJB. Un crash è trasparente al client.

javax.ejb.SessionBean è un oggetto di tipo sessione con le seguenti caratteristiche:

1. Esegue operazioni in nome di un singolo client.
2. Può essere una transazione.
3. Aggiorna dati condivisi in un database sottostante. Non rappresenta direttamente i dati condivisi del database, sebbene possa accedere e aggiornare tali dati.
4. Ha una vita relativamente breve.
5. Viene rimosso quando il server EJB subisce un crash. Il client deve ristabilire un nuovo oggetto session per continuare l'operazione.

8.3 Entity Bean

Un entity bean rappresenta un oggetto view di dati memorizzati in una memoria persistente o in un'applicazione esistente. Fornisce un involucro attorno ai dati per semplificare il processo di accesso e manipolazione agli stessi. L'interfaccia di questo oggetto si presta alla riusabilità del software (si pensi ad un entity bean che rappresenta le informazioni sull'account di un utente). Un entity bean permette accesso condiviso da client multipli e può vivere oltre la durata della sessione di un client con il server. Se lo stato di un entity bean stava per essere aggiornato da una transazione al momento del crash del server, tale stato è automaticamente resettato allo stato dell'ultima transazione di cui è stato fatto il commit.

Riassumendo, un entity bean rappresenta quindi un oggetto **persistente** e consente di mappare una sorgente dati su una classe Java (tabella, vista, join o stored procedure in un database relazionale, dati legacy opportunamente incapsulati). Ciascuna istanza può essere condivisa da tanti client. Quando l'istanza di un entity bean smette di esistere, il dato che essa rappresenta continua ad esistere sul database. Importante è sottolineare che in caso di crash del sistema i dati rappresentati dagli entity bean resistono, mentre i session bean vengono perduti.

Un entity bean fornisce anche i metodi per agire sui dati che esso stesso rappresenta. Ciascuna istanza di un entity bean è identificata in modo univoco da una chiave primaria (rappresentata da uno o più campi della tabella): la **primary key**.

8.3.1 Linee guida all'uso degli Entity Bean

Quando un business object deve essere modellato come un entity bean?

- **Rappresentare dati persistenti**: se lo stato di un business object deve essere memorizzato in una memoria persistente ed il suo behavior rappresenta principalmente la manipolazione di dati rappresentati nel suo stato.
- **Fornire accessi concorrenti da molteplici client**: quando lo stato ed il behavior di un business object devono essere condivisi tra molteplici client e lo stato mantenuto tra successive chiamate a metodo.
- **Rappresentare un singolo record (riga) logico**: se il business object opera su un unico record logico di un database (gli entity bean forniscono metodi per individuare, creare e manipolare una riga alla volta).
- **Fornire una robusta e long-lived gestione di dati persistenti**: se un business object deve vivere dopo che una sessione client con il server è terminata oppure deve essere presente quando il server viene riavviato dopo il crash.

8.3.2 Esempio: uno User Account Bean

Il concetto di account utente è centrale nella maggior parte delle applicazioni di e-commerce. Molteplici clienti hanno bisogno di condividere dei behavior come la creazione di un account, la

verifica di un account esistente, l'aggiornamento delle informazioni relative ad un account. Gli aggiornamenti allo stato di un oggetto account devono essere scritti in una memoria persistente ed un oggetto account vive anche quando la sessione client con il server è terminata. Dunque un tale oggetto può essere modellato come un entity bean. Il codice di esempio mostra l'interfaccia dell'enterprise bean Account e l'implementazione di AccountDetails.

```
public interface Account extends EJBObject {
    public void changeContactInformation (ContactInformation info) throws RemoteException;
    public AccountDetails getAccountDetails ( ) throws RemoteException;
}

public class AccountDetails implements java.io.Serializable {
    private String userId;
    private String status;
    private ContactInformation info;
    public String getUserId ( ) {
        return userId;
    }
    ...
}
```

Esempio 8.1 L'interfaccia remota Account e la classe AccountDetails

8.3.3 La persistenza dei dati

La persistenza dei dati rappresentati da un entity bean può essere a carico del bean (Bean-Managed Persistence o BMP) o del container (Container-Managed Persistence o CMP).

Nel caso di **Bean-Managed Persistence**, l'interfacciamento/mapping con le tabelle del database è cablato all'interno del codice dell'enterprise bean. Lo sviluppatore implementa le istruzioni di accesso al database (ejbCreate(), ejbLoad(), ejbStore(), ejbRemove()) utilizzando JDBC. Risulta però difficile il supporto per eventuali caratteristiche avanzate offerte dal container (connection pooling e data caching). Inoltre è facile che si rendano necessarie frequenti modifiche al codice sorgente dell'EJB in caso di modifiche nello schema del database, cambiamento del DB vendor (ad es. da SQL Server ad Oracle), cambiamento della tipologia di sorgente (ad es. da RDBMS ad OODBMS). Si ha un piccolo vantaggio nell'eventuale ottimizzazione delle tabelle sul database.

Nel caso di **Container-Managed Persistence** l'accesso al database è gestito completamente dal container, il quale genera automaticamente il codice SQL per l'accesso alle tabelle sul database. L'accesso ai dati del bean da parte dei client si traduce in opportune query sul database. La modifica dei dati contenuti nel bean (setProperty()) si traduce in opportune update sul database. Come risultato si ha sviluppo degli entity bean quasi totalmente automatico e flessibilità maggiore rispetto alla sorgente dati. In questo caso quindi gli enterprise bean sono più robusti: il minor ricorso a JDBC implica un gran numero di righe di codice in meno, cioè statisticamente meno errori. Lo svantaggio risiede nel fatto che il container è più complesso e dunque più costoso. Inoltre non sempre è possibile far gestire in automatico l'accesso ai dati.

Da questa breve trattazione risulta che, quando è possibile, è consigliabile usare la Container-Managed Persistence.

8.4 Session Bean

I session bean sono usati per implementare business object che mantengono una business logic client-specific. Lo stato di un session bean riflette le sue interazioni con un particolare client e non è inteso per accesso generale. Dunque, un session bean tipicamente esegue per conto di un singolo client e non può essere condiviso tra molteplici client. Un session bean è un'estensione logica del programma client che gira sul server e contiene informazioni specifiche al client. Al contrario degli entity bean, i session bean non rappresentano direttamente dati del database condivisi, anche se possono accedere ed aggiornare tali dati. Lo stato di un session bean è **non-persistente** e non deve essere scritto nel database.

Un session bean non è quindi persistente. Implementa l'interfaccia **javax.ejb.SessionBean**. In genere implementa la logica di business dell'applicazione: riceve le invocazioni dal client ed agisce

(creazione, modifica, cancellazione) sugli entity bean. Dunque un session bean si comporta come client verso gli entity bean e l'accesso agli entity bean risulta così maggiormente protetto. La sequenza delle operazioni è eseguita in genere all'interno di una transazione.

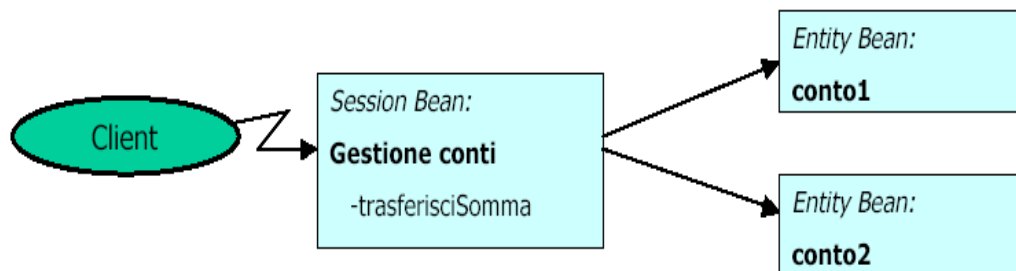


Figura 8.5 Esempio di utilizzo di un session bean

Ciascuna istanza è da considerarsi una estensione logica dell'applicazione client. La vita di un session bean è legata al client che l'ha istanziato: un session bean è come un agente dedicato allo specifico client, in esecuzione sul server.

Un session bean è generalmente stateful. Comunque, le specifiche EJBs descrivono stateless session bean per fornire server-side behavior che non mantiene alcuno stato specifico. Dunque esistono **stateful session bean** e **stateless session bean**.

8.4.1 Stateful Session Bean

Uno stateful session bean mantiene uno stato conversazionale per conto del client. Uno stato conversazionale è definito dai valori dei campi del session bean più tutti gli oggetti raggiungibili da tali campi. Un session bean non rappresenta direttamente dati in una memoria persistente, ma può accedere ed aggiornare i dati per conto del client. Il lifetime di uno stateful session bean è tipicamente quello del suo client. Un business object deve essere modellato come uno stateful session bean per:

- Mantenere uno stato client-specific
- Rappresentare oggetti non persistenti
- Rappresentare work flow tra i business object

Nel caso di utilizzo di uno stateful session bean viene creata una sessione 1:1 con uno specifico client. Viene mantenuto lo stato tra una invocazione di un metodo e l'altra. Quando il client si disconnette, il bean viene automaticamente rimosso. In caso di inattività del client, il container può scegliere di passivare il componente. Il componente può essere in seguito riattivato.

8.4.1.1 Esempio: uno Shopping Cart Bean

Uno shopping cart object rappresenta la collezione di prodotti selezionati da un particolare utente per l'acquisto durante una sessione. Lo stato di un tale oggetto è specifico di una particolare sessione utente e non ha bisogno di essere salvato, a meno che l'utente sia pronto all'ordine. L'oggetto è short-lived. I dati non dovrebbero essere condivisi, dato che rappresentano una particolare interazione con uno specifico utente e vivono solo per la sessione utente con il server.

```

public interface ShoppingClientController extends EJBObject {
    public Catalog getCatalog ( ) throws RemoteException;
    public ShoppingCart getShoppingCart ( ) throws RemoteException;
    public Account getAccount ( ) throws RemoteException;
    public Collection getOrders ( ) throws RemoteException, FinderException;
    public Order getOrder (int requestId) throws RemoteException, FinderException;
    // Returns a list of updated models
    public Collection handleEvent (EStoreEvent se) throws RemoteException, DuplicateAccountException;
}
  
```

Esempio 8.2 L'interfaccia remota ShoppingClientController

8.4.2 Stateless Session Bean

Gli stateless session bean sono disegnati rigorosamente per fornire server-side behavior. Sono anonimi, nel senso che non contengono alcun dato user-specific. Quando modellare un business object come stateless session bean?

- Modellare oggetti riusabili
- Fornire elevate performance
- Operare contemporaneamente su multiple righe
- Fornire una vista procedurale dei dati

Nel caso di impiego di uno stateless session bean non viene mantenuto alcuno stato tra un'invocazione di un metodo e l'altra. Una stessa istanza del bean può servire più client contemporaneamente; ciò implica ovviamente maggiori performance.

8.4.2.1 Esempio: un Catalog Bean

Un catalog object rappresenta differenti categorie e prodotti e fornisce i servizi di navigazione e ricerca ai suoi client. Entrambe queste due funzioni primarie sono servizi generici che non sono legati ad alcun client particolare. Inoltre, tale oggetto opera contemporaneamente su multiple righe di un database e fornisce una vista condivisa dei dati. Il codice dell'esempio lista i servizi forniti da un catalog object:

```
public interface Catalog extends EJBObject {
    public Collection getCategories ( ) throws RemoteException;
    public Collection getProducts (String categoryId, int startIndex, int count) throws RemoteException;
    public Product getProduct (String productId) throws RemoteException;
    public Collection.getItems (String productId, int startIndex, int count) throws RemoteException;
    public Item getItem (String itemId) throws RemoteException;
    public Collection searchProducts (Collection keyWords, int startIndex, int count) throws RemoteException;
}
```

Esempio 8.3 L'interfaccia remota Catalog

Un altro esempio di stateless session bean è il mailer object usato per spedire mail di conferma a clienti dopo che il loro ordine è stato passato con successo. Il mailer fornisce un generico servizio che può essere completato entro una singola chiamata a metodo; il suo stato non è legato a nessun client particolare.

Inoltre, dato che le istanze possono essere condivise tra molteplici client, sono modellate come stateless session bean.

```
public Collection getProducts (String categoryId, int startIndex, int count) {
    Connection con = getDBConnection( );
    try {
        CatalogDAO dao = new CatalogDAO (con);
        return dao.getProducts (categoryId, startIndex, count);
    } catch (SQLException se) {
        throw new GeneralFailureException(se);
    } finally {
        try {
            con.close();
        } catch (Exception ex) {
            ...
        }
    }
}
```

Esempio 8.4 CatalogImpl.getProducts

```

public Collection getProducts (String categoryId, int startIndex, int count) throws SQLException {
    String qstr = "SELECT itemid, listprice, unitcost, attr1, a.productid, name, descn " +
        "FROM item a, product b WHERE " +
        "a.productid = b.productid AND category = '" + categoryId + "' ORDER by name";
    ArrayList al = new ArrayList ( );
    Statement stmt = con.createStatement( );
    ResultSet rs = stmt.executeQuery (qstr);
    HashMap table = new HashMap ( );
    // skip initial rows as specified by the startIndex parameter
    while (startIndex-- > 0 && rs.next( ));
    // Now get data as requested
    while (count-- > 0 && rs.next( )) {
        int i = 1;
        String itemid = rs.getString (i++).trim( );
        double listprice = rs.getDouble (i++);
        double unitcost = rs.getDouble (i++);
        ...
        Product product = null;
        if (table.get (productid) == null) {
            product = new Product (productid, name, descn);
            table.put (productid, product);
            al.add (product);
        }
    }
    rs.close();
    stmt.close();
    return al;
}

```

Esempio 8.5 CatalogDAO.getProducts

8.5 Riassunto

Esistono diversi servizi comuni che sono richiesti dalle applicazioni enterprise distribuite. Questi includono mantenere lo stato, operare su dati condivisi, partecipare a transazioni, servire un largo numero di client, fornire accesso remoto ai dati, e controllare l'accesso ai dati. Il Middle Tier si è sviluppato come il luogo ideale per fornire questi servizi. J2EE promuove l'architettura Enterprise JavaBeans come il modo per fornire i servizi di sistema necessari a gran parte delle applicazioni. Tale architettura libera gli sviluppatori dal doversi occupare di questi servizi permettendo loro di concentrarsi sulla business logic.

L'architettura EJB fornisce vari tipi di enterprise bean per modellare business object: entity bean, stateful session bean e stateless session bean. La scelta di un particolare enterprise bean per modellare un business concept dipende da diversi fattori tra i quali la necessità di fornire una gestione robusta dei dati, di fornire behavior efficienti, e di mantenere lo stato del client durante una sessione utente.

Un entity bean fornisce una vista object-oriented dei dati memorizzati in un database; uno stateless session bean dà una vista procedurale dei dati. Un Application Component Provider dovrebbe usare entity bean per modellare entità logiche come record singoli di un database. Nell'implementare il behavior per visitare righe multiple in un database e presentare una vista read-only dei dati, gli stateless session bean sono la scelta migliore. Essi sono disegnati per fornire servizi di accesso generico a molteplici client.

Alcuni business concept richiedono qualcosa di più di una view dei dati. Un esempio potrebbe essere un catalogo, che fornisce servizi di navigazione e ricerca così come meccanismi per aggiornare le informazioni sul prodotto. In tali casi, si può usare uno stateless session bean per operare sulle informazioni di un prodotto nell'insieme e un entity bean per fornire accesso ad un particolare prodotto.

Dato che gli enterprise bean sono oggetti remoti che consumano una significativa quantità di risorse di sistema e di larghezza di banda di rete, essi non sono appropriati per modellare qualsiasi business object. Un Application Component Provider può usare Data Access Objects per

incapsulare l'accesso ad un database e value object (oggetti Java serializzabili che possono essere passati al client per valore) per modellare oggetti che sono dipendenti dagli enterprise bean.

Inoltre, può non essere appropriato dare ai client accesso diretto a tutti gli enterprise bean usati dall'applicazione. Alcuni enterprise bean possono agire come mediatori per la comunicazione tra client ed EJB Tier. Tali bean possono incapsulare specifici work flow per un'applicazione, o servire come entry point per una gerarchia di informazioni.

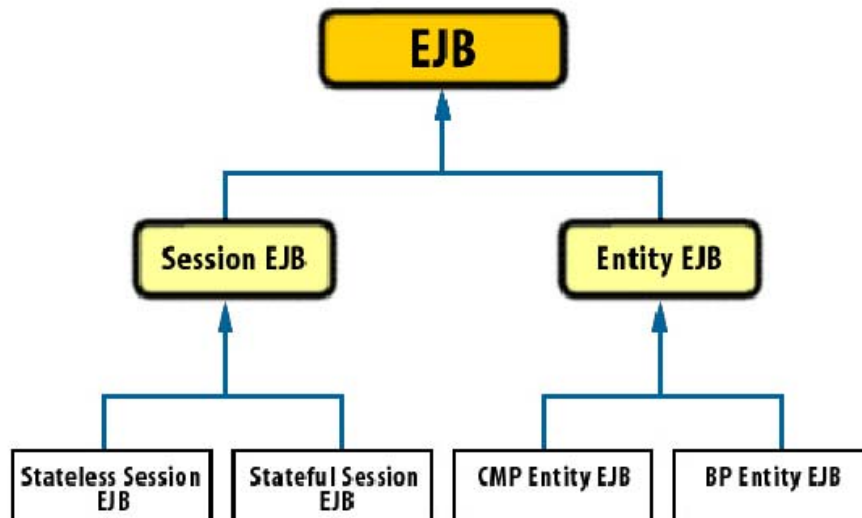


Figura 8.6 I componenti EJB

Capitolo 9 L'Enterprise Information System Tier

Le applicazioni enterprise richiedono (per loro definizione) accesso ad altre applicazioni, dati o servizi sparsi all'interno di diversi EISs. Questi sistemi forniscono l'infrastruttura informatica per le informazioni di un'azienda. Esempi di EISs includono:

- Sistemi Enterprise Resource Planning (ERP)
- Sistemi Mainframe Transaction Processing
- Relational DBMS (RDBMS)
- Applicazioni legacy
- Non-relational DBMS (es. OODBMS)

Le aziende operano usando le informazioni memorizzate in questi sistemi: il successo di un'azienda dipende in maniera critica da queste informazioni, che ne rappresentano dunque la ricchezza, e come tale vanno trattate con estrema cura. Un'azienda non può permettersi che un'applicazione causi inconsistenza nei dati o comprometta l'integrità degli stessi. Questo conduce ad un requisito fondamentale: assicurare accessi transazionali agli EISs da varie applicazioni.

Il modello bidimensionale delle vecchie forme di business legato ai sistemi informativi è stato sostituito da un nuovo modello (**e-business**) che introduce una terza dimensione a rappresentare la necessità di garantire **accesso ai dati contenuti nella infrastruttura enterprise via Web** a partner commerciali, clienti, impiegati e altri sistemi informativi.

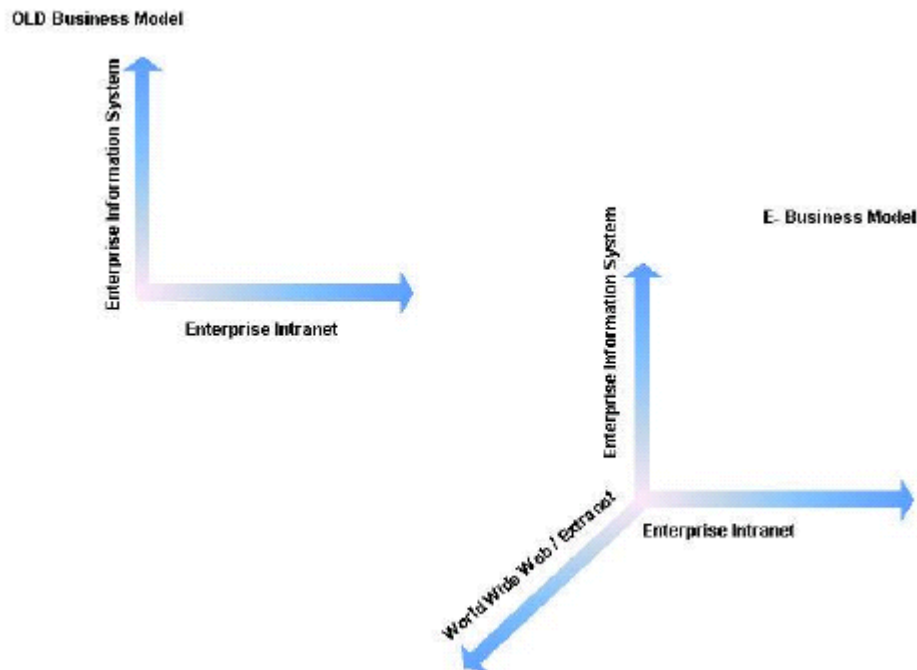


Figura 9.1 Old Business Model vs. E-business Model

Tipicamente le aziende sviluppano applicazioni Web che accedono e gestiscono informazioni memorizzate nei loro sistemi informativi. Sviluppando tali applicazioni come applicazioni J2EE si possono raggiungere facilmente i requisiti di sicurezza, scalabilità e transazionalità.

Gli scenari di riferimento sono quindi svariati e comprendono vari modelli di configurazione che le architetture enterprise vanno ad assumere per soddisfare le necessità della new economy. Un modello classico è quello rappresentato da sistemi di commercio elettronico. Nei negozi virtuali (o E-Store) l'utente Web interagisce tramite browser con i cataloghi on-line del fornitore, seleziona i prodotti, inserisce i prodotti selezionati nel carrello virtuale, avvia una transazione di pagamento con protocolli sicuri. Esistono innumerevoli configurazioni nelle quali una applicazione J2EE deve essere strutturata per accedere ad un EIS. Di seguito alcuni esempi di scenari di integrazione di un'applicazione con l'EIS.

9.1 Una applicazione Internet E-Store

La compagnia A gestisce una applicazione di e-store basata sulla piattaforma J2EE. Questa applicazione è composta da un insieme di enterprise bean, pagine JSP, e servlet che collaborano per fornire tutte le funzionalità dell'applicazione. Il database memorizza dati che riguardano il catalogo dei prodotti, il carrello della spesa (shopping cart), la registrazione ed i profili dei clienti, lo stato e le registrazioni di una transazione. L'architettura di questa applicazione è illustrata in Figura 9.2.

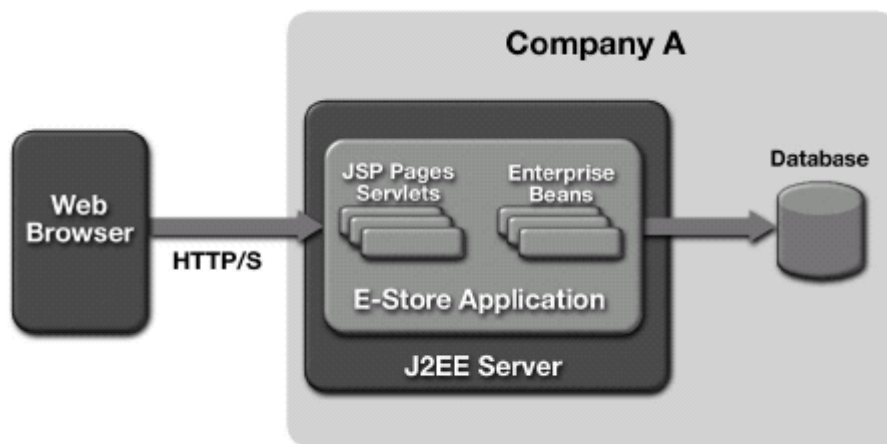


Figura 9.2 Un'applicazione Internet E-Store

Un cliente usa un browser Web per avviare una transazione di e-commerce con l'applicazione di e-store. Un cliente naviga il catalogo, fa una selezione di prodotti, inserisce tale selezione in un carrello virtuale, digita un username ed una password per avviare una transazione di pagamento sicura, completa un modulo riguardante informazioni sull'ordine, ed infine conferma l'ordine. In tale scenario, l'applicazione usa un database esistente che contiene già le informazioni su prodotti ed inventario per memorizzare tutte le informazioni persistenti riguardanti i clienti e le loro transazioni.

9.2 Una applicazione Intranet Human Resources

L'applicazione Intranet per la gestione delle risorse umane della compagnia B proposta per questo scenario supporta un'interfaccia Web alle applicazioni Human Resources (HR) esistenti supportate dal sistema Enterprise Resource Planning del vendor X e fornisce processi business aggiuntivi customizzati sui bisogni della compagnia B.

La Figura 9.3 illustra un'ipotetica architettura per questa applicazione. Il J2EE Middle Tier è composto da enterprise bean, pagine JSP e servlet che forniscono personalizzazione dei processi business e supportano una interfaccia Web standardizzata. Tale applicazione permette l'accesso ad un qualsiasi impiegato della compagnia B.

Le funzionalità offerte sono diverse, dipendentemente dal ruolo (Manager, HR manager, e Employee) con cui l'impiegato accede. L'applicazione permette di realizzare varie funzioni di gestione del personale: gestione delle informazioni sul personale, gestione delle paghe, gestione dei risarcimenti, amministrazione dei benefit, gestione dei viaggi, pianificazione dei costi.

Il dipartimento IT della compagnia B è responsabile del deployment dell'applicazione. L'accesso all'applicazione è permesso solo agli impiegati della compagnia operanti all'interno della Intranet aziendale e si basa sui loro ruoli e privilegi di accesso.

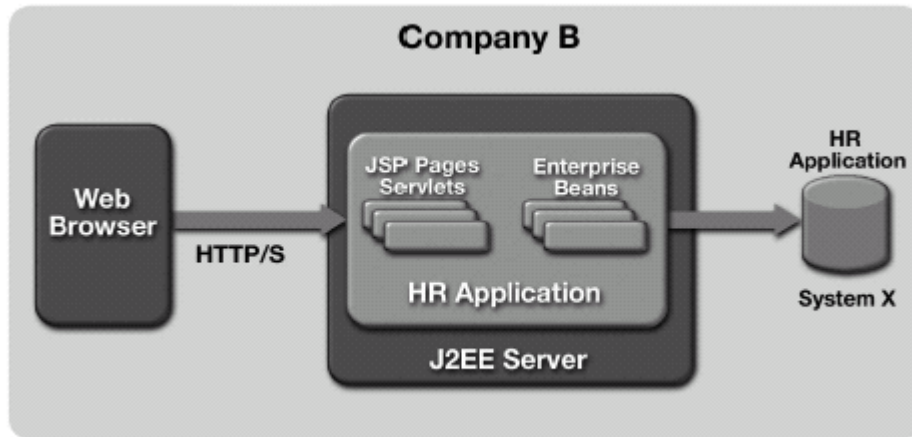


Figura 9.3 Un'applicazione Intranet Human Resources

9.3 Una applicazione per il Distributed Purchasing

Questa applicazione permette ad un impiegato della compagnia C di usare un'interfaccia Web-based per realizzare transazioni di acquisto multiple. Un impiegato può gestire l'intero processo di procurement, dalla creazione di una richiesta di acquisto all'ottenimento dell'approvazione della fattura. Questa applicazione inoltre si integra con le applicazioni finanziarie esistenti nell'azienda per tenere traccia degli aspetti finanziari del processo di procurement.

La Figura 9.4 illustra un'architettura per questa applicazione. L'applicazione è composta da pagine JSP, servlet, enterprise bean e sistemi informativi esistenti. Gli enterprise bean integrano un'applicazione di logistica, fornita dal vendor X, che fornisce funzioni integrate di gestione dell'acquisto e dell'inventario, ed un'altra applicazione, sviluppata dal vendor Y, che fornisce financial accounting (contabilità finanziaria).

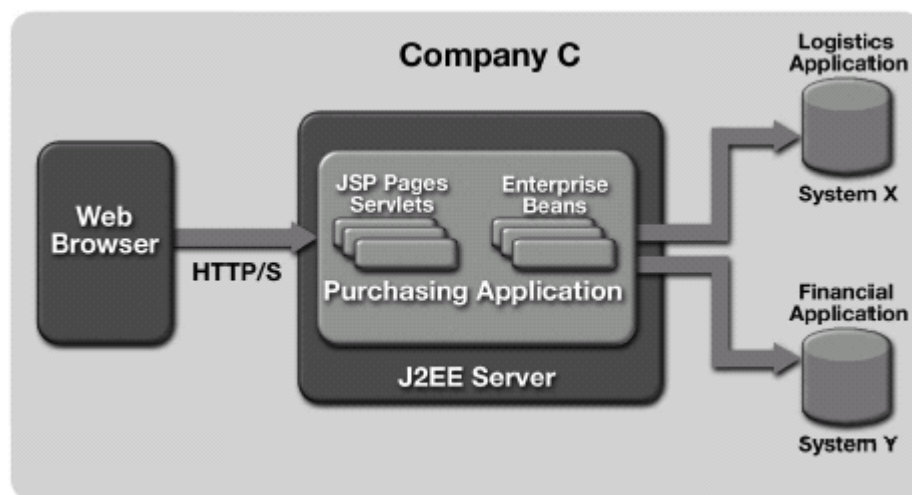


Figura 9.4 Una Distributed Purchasing Application

In questo scenario, la compagnia C è un'azienda largamente decentralizzata, con unità e dipartimenti geograficamente distribuiti. Il sistema X ed il sistema Y sono gestiti da differenti dipartimenti IT e sono stati deployati in location geografiche differenti. L'applicazione di acquisto integrata è deployata in una location diversa sia da quella di X che di Y.

Le applicazioni X e Y fanno parte di differenti domini di sicurezza; usano differenti tecnologie per implementare la sicurezza ed hanno i loro specifici meccanismi e politiche di sicurezza. L'applicazione di acquisto distribuito (distributed purchasing application) è deployata in un dominio di sicurezza diverso sia da quello di X che da quello di Y.

Capitolo 10 Un approfondimento sugli Application Server

Il mercato degli Application Server si sta dimostrando essere uno dei settori a più alta crescita tra tutti i settori tecnologici del software. Un mercato, tuttavia, per il quale non esiste ancora una definizione unica e accettata. Ciascun produttore ha coniato una propria definizione, forte del fatto che chiunque si affacci al mondo Internet e all'Electronic Business non può pensare di farne a meno. La maggior parte delle aziende che producono strumenti di sviluppo software e dei produttori di database, infatti, sono ormai entrati nel mercato degli Application Server, da molti considerato come l'area dotata delle migliori prospettive di crescita a partire dall'avvento dei database relazionali. Ad accentuare la confusione contribuiscono da un lato gli utenti che "hanno sempre più fretta" di affacciarsi sulla rete, dall'altro la complessità della tecnologia e la nascita di nuovi standard e di modelli operativi che mettono in discussione ogni certezza e creano ampie opportunità ad un'intera generazione di aziende create apposta per la Web Economy. Come è già successo nel caso di altri prodotti "esplosivi", all'inizio la confusione regna sovrana: un rapido sguardo al mondo del software rivela che ci sono più di 25 società che si fregiano della qualifica di produttori di Application Server. Uno dei primi problemi nell'affrontare il mondo degli Application Server consiste quindi nel darne una definizione sufficientemente precisa da identificare le loro funzioni.

Un **Application Server** è un software che fornisce i servizi necessari a supportare applicazioni Web-based che consentono agli utenti di accedere ai database aziendali. Un Application Server, quindi, agisce in un certo senso da intermediario fra i client (spesso, ma non necessariamente, browser Web) e i database server, evitando alle aziende di dover installare sulle macchine client complesse applicazioni client/server, le quali normalmente richiedono un elevato impegno in fatto di manutenzione e quindi di costo.

Da questa breve definizione traspare l'organizzazione a tre livelli tipica di un'infrastruttura basata su Application Server, che può essere vista come un'evoluzione del tradizionale modello client/server, e costituita dai client che normalmente sono dotati di un browser e non hanno intelligenza per il dialogo diretto con il back-end (es. un sistema legacy), uno o più Application Server che agiscono appunto da intermediari fra il client e il back-end, e uno o più data server.

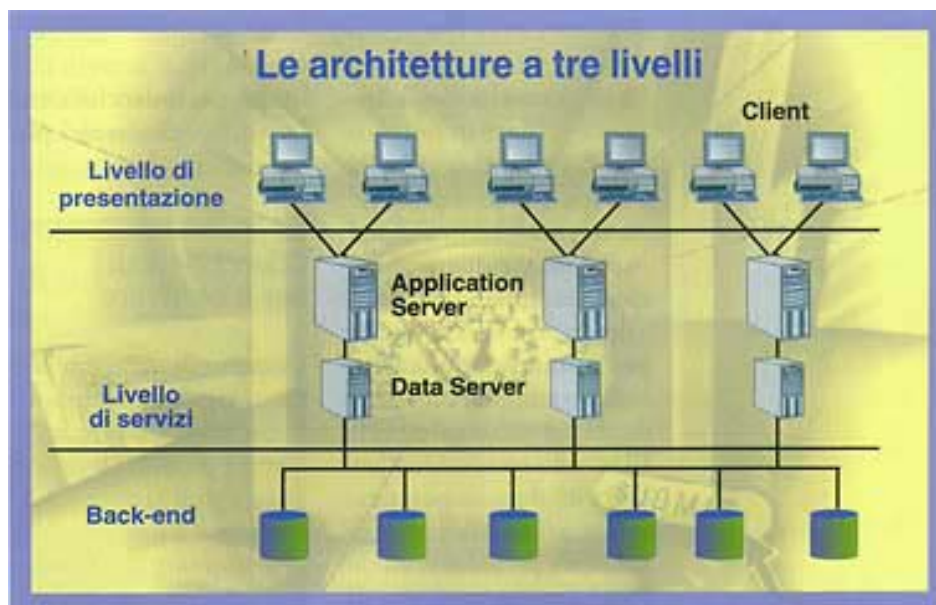


Figura 10.1 Architettura a tre livelli tipica di un'infrastruttura basata su Application Server

Quando si vuole utilizzare il Web per permettere agli utenti interni, oppure a clienti e partner che operano all'esterno via Internet, di consultare le informazioni aziendali, si vuole anche che le operazioni risultino estremamente semplici, e che l'accesso ai database non sia concesso a tutti indiscriminatamente. L'allargamento nella fruizione e nell'accesso ai dati deve essere sicuro e controllato: questa è la tipica situazione in cui, normalmente, entra in scena un Application

Server. Questi pacchetti software, tipicamente scritti in linguaggio Java per poter essere utilizzati su piattaforme diverse, costituiscono quindi il collegamento tra gli utenti finali che utilizzano un browser e i database contenenti le informazioni alle quali essi devono accedere. Un Application Server gestisce gran parte del processo di connessione degli utenti verso i dati e viceversa: crea una sessione di applicazione per ogni utente, lo identifica, estrae i dati richiesti dagli archivi opportuni, li elabora e, infine, li visualizza direttamente al client creando una pagina Web.

Questo software fornisce tutto ciò che serve al migliore collegamento tra la pagina del browser Web e ciò che si trova all'interno del database aziendale rendendo disponibili via Internet/Intranet le informazioni elaborate, su client di qualsiasi tipo, purché dotati di un browser. Questo significa che non è necessario installare prodotti particolari sui PC degli utenti finali, rendendo possibile lo sviluppo di applicazioni di tipo thin client, ovvero software basato su server che viene eseguito su browser Web o terminali dedicati e che richiede un supporto desktop minimo.

L'obiettivo è "consentire di **implementare architetture thin client ovunque**". Tutto il codice necessario per eseguire un'applicazione viene fondamentalmente scritto nel software del server, fatto che semplifica notevolmente aspetti di manutenzione e aggiornamento delle applicazioni.

Il software pacchettizzato per Application Server, in vendita oggi da parte di diversi produttori, gran parte dei quali hanno meno di cinque anni di vita ed altissimi tassi di sviluppo, fornisce anche servizi vitali di gestione delle applicazioni, come il monitoraggio delle prestazioni del sistema, al fine di prevedere ed evitare i colli di bottiglia. La gestione delle prestazioni è inoltre fondamentale su applicazioni che, accessibili da tutta l'azienda, o per assurdo dal mondo intero (attraverso Internet), devono poter essere in grado di operare in tempi accettabili. Questi servizi possono inoltre essere legati insieme per formare applicazioni su larga scala, che richiedono la presenza di più server per soddisfare tutti le necessità degli utenti in termini di informazioni e dati.

Le aziende che vogliono realizzare applicazioni sempre più complesse, possono legare gli Application Server a prodotti software basati su server separati, i quali elaborano gli ordini e altre transazioni distribuendo parti di codice riutilizzabile (oggetti) agli utenti.

10.1 Gli elementi essenziali di un Application Server

Più in dettaglio, un Application Server include alcuni elementi essenziali:

- gli strumenti per realizzare applicazioni sotto forma di componenti software
- un insieme di programmi che girano sul server per eseguire e gestire tali componenti
- i moduli per interfacciarsi al sistema informativo (client, sorgenti dei dati e così via) in cui si va a calare l'Application Server

Ciò ha richiesto in passato lo sviluppo di codice compatibile con le diverse API dei vari Application Server; ora buona parte dei produttori si sta uniformando alle specifiche Enterprise JavaBeans di Sun. Gli EJB (o enterprise bean) sono componenti software aderenti alle API di J2EE la quale, a sua volta, è la piattaforma Java che Sun propone al mondo enterprise.

Oltre a Java, molti Application Server supportano anche altri linguaggi di programmazione e sono coadiuvati da ambienti di sviluppo per la generazione della business logic. L'offerta di ulteriori moduli supplementari varia da produttore a produttore, ma generalmente tutti gli Application Server più recenti sono dotati anche di connettori predefiniti per interfacciarsi con le applicazioni più diffuse in ambito aziendale, per esempio quelle di Peoplesoft o SAP. Anche il dialogo con i sistemi e gli archivi legacy è un aspetto cui i vendor pongono notevole attenzione; questo permette di utilizzare gli Application Server come mezzo per "ringiovanire", almeno nei confronti dei client, molti sistemi informativi preesistenti portandoli verso il Web. In realtà particolarmente eterogenee, inoltre, un Application Server può servire per coordinare i flussi di lavoro delle diverse applicazioni con cui si interfaccia, tanto da spingere alcuni analisti a ritenere che gli Application Server più sofisticati potranno diventare il mezzo attraverso cui le aziende permetteranno (o vieteranno) l'accesso ai propri processi di business interni a entità esterne, come i fornitori, i clienti o i partner.

10.2 Java e i server, una coppia ideale

Come piattaforma, Java ha nel tempo sviluppato una serie di caratteristiche che possono avvantaggiarsi dell'elaborazione lato server. Tra queste:

- L'API **JDBC**, che interfaccia il software Java con i database.
- I **servlet** (piccole applicazioni Java server-side), che interfacciano i browser con i servizi Web in maniera migliore rispetto alle convenzionali CGI.
- La tecnologia **JSP** che, come ASP di Microsoft, integra moduli di codice nelle pagine HTML.
- Gli **Enterprise JavaBeans**, i quali definiscono un'architettura standard a componenti per la condivisione di codice Java.

Gli Application Server Java hanno inoltre risolto alcuni dei problemi che avevano limitato Java come tecnologia server, tra i quali:

- La **gestione delle sessioni**: il Web è un ambiente definito stateless (senza stato), ma un servizio di rete deve sapere a quale utente sta parlando. Gli Application Server creano e mantengono le sessioni utente.
- La **sicurezza**: i servizi di rete devono sapere quali utenti possono visualizzare certe pagine e gli Application Server possono in questo senso definire e applicare procedure di sicurezza.
- Il **collegamento ad alte prestazioni verso i database**: un Application Server può definire una serie di connessioni a un database e poi mantenerle aperte, per riutilizzarle e garantire un accesso veloce ai dati.
- La **fault tolerance**: un servizio di rete non dovrebbe bloccarsi; gli Application Server possono distribuire i servizi tra più JVM e CPU per una maggiore sicurezza.

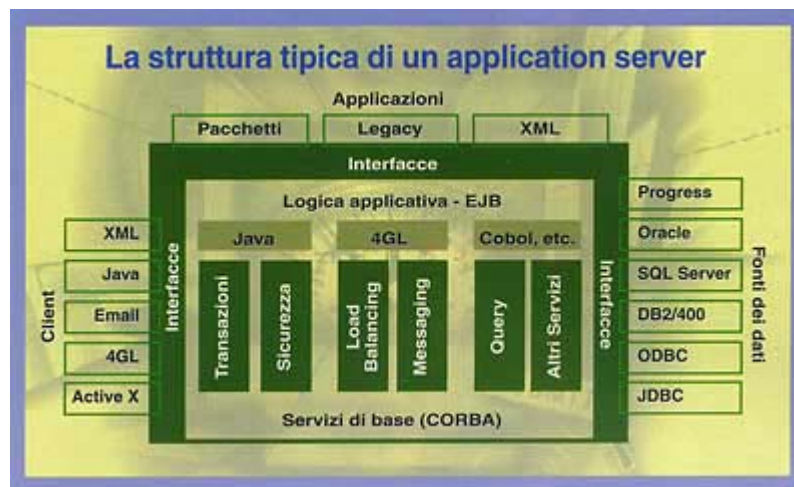


Figura 10.2 La struttura tipica di un Application Server

Nonostante sia stato pubblicizzato soprattutto per quello che riesce (o non riesce) a fare all'interno di un browser Web, Java ha trovato sul lato server le sue migliori applicazioni, proponendosi come linguaggio di sviluppo e piattaforma preferenziale per realizzare applicazioni Intranet e mission-critical. Java ha, infatti, una serie di caratteristiche che lo rendono indicato per lo sviluppo di servizi di rete e, per di più, l'ambiente server tende a ridurre al minimo le sue debolezze. I tempi interminabili di scaricamento ed esecuzione che molti utenti devono attendere quando accedono ad applicazioni Java residenti sui siti Internet, per esempio, non si registrano in architetture basate su Application Server, perché qui i servizi Java sono sempre disponibili una volta attivato l'Application Server stesso. Dato che gli Application Server fondamentalmente si "esprimono" in HTML, non sussistono inoltre i problemi di compatibilità fra le diverse Java Virtual Machine, problemi che talvolta colpiscono i siti Web.

Molti IT manager ritengono che con la piattaforma J2EE Java abbia compiuto un importante passo necessario per diventare una piattaforma per la realizzazione di applicazioni aziendali su larga scala, questo anche grazie all'approccio modulare degli Enterprise JavaBeans.

10.3 Perché l'Application Server? Vantaggi

L'adozione degli Application Server offre benefici alle aziende che progettano applicazioni Web-based, soprattutto nelle aree dello sviluppo, dell'esecuzione e della gestione integrata dei sistemi.

I principali vantaggi apportati nell'ambito dello sviluppo possono essere così riassunti:

■ **Semplificazione delle attività di sviluppo.** Normalmente, gli Application Server creano un ambiente nel quale si possono utilizzare gli strumenti di sviluppo più diffusi sul mercato, consentendo di produrre e distribuire rapidamente applicazioni transazionali altamente scalabili. In generale, questi ambienti comprendono delle ricche librerie C, C++ e Java, modelli e strumenti di ausilio per sviluppare le applicazioni e pacchetti specifici predefiniti che riducono i tempi di realizzazione e messa in esercizio dei programmi negli ambienti distribuiti.

■ **Supporto di vari linguaggi, strumenti e piattaforme.** Gli strumenti di sviluppo dovrebbero assicurare massima flessibilità e totale libertà di scelta: le applicazioni possono essere scritte nel linguaggio preferito dal programmatore, inclusi Java, C e C++. Inoltre, devono permettere l'integrazione con altri strumenti di authoring HTML, editor, compilatori e con il software di gestione delle comunicazioni. Le varie piattaforme supportate devono comprendere UNIX e Windows NT.

■ **Riusabilità del codice.** Una volta sviluppata, la logica applicativa può essere condivisa e riutilizzata, consentendo ai programmatori il futuro potenziamento dei propri applicativi.

I vantaggi introdotti da un Application Server in fase di elaborazione sono i seguenti:

■ **Gestione delle richieste e delle transazioni.** Grazie alla funzione di gestione degli stati e delle sessioni, l'Application Server supporta le richieste, complesse e multistep, che intervengono nelle operazioni basate su transazioni. Il gestore delle transazioni e quello delle richieste devono assicurare integrità transazionale e gestione affidabile dei back-end multipli per le risorse e i dati. Il transaction manager gestisce le interazioni con i database e le funzioni di Commit, Rollback e Recovery, a fronte di ogni possibile malfunzionamento. L'accesso comune ai dati e una serie di API che gestiscono le transazioni sono in grado di unificare l'accesso a sistemi composti da database eterogenei.

■ **Scalabilità.** Gli Application Server supportano il partizionamento delle applicazioni e forniscono un'architettura orientata alla massima scalabilità. Ogni istanza dell'AS riesce a elaborare un alto volume di transazioni; i sistemi multi-CPU e i cluster di Application Server assicurano la scalabilità necessaria a gestire anche grandi quantità di utenti concorrenti. Questi obiettivi vengono raggiunti attraverso l'impiego di adeguate piattaforme hardware, sistemi operativi e opportune configurazioni dei sistemi. Grazie all'architettura scalabile degli Application Server, gli sviluppatori possono progettare le applicazioni per soddisfare le richieste del mercato. La scalabilità lineare, cioè il processo che consiste nell'ottenere prestazioni maggiori in proporzione all'adozione di piattaforme più potenti, rende gli Application Server una soluzione ottimale per gestire applicazioni che, partendo da modesti numeri di utenti e transazioni, debbono poter arrivare a supportare grandi volumi di transazioni.

■ **Alte prestazioni.** Gli Application Server offrono delle nuove caratteristiche architettoniche che assicurano la capacità di erogare elevate prestazioni quali il multithreading, la gestione degli oggetti e il bilanciamento dinamico dei carichi di lavoro a livello del kernel. Gli AS consentono inoltre di accedere ad alta velocità ai dati, garantendo rapidi tempi di risposta anche quando sono sottoposti a pesanti carichi di lavoro: questo è possibile grazie ai servizi di sistema quali la gestione thread-pool, lo streaming, il caching delle connessioni ai database e quello dei risultati dinamici. Gli Application Server comunicano con i Web server tramite script CGI, NSAPI e ISAPI ottimizzati: queste interfacce garantiscono un significativo aumento delle prestazioni mantenendo l'indipendenza dal Web server. Grazie a queste interfacce, gli sviluppatori non devono più preoccuparsi della dipendenza del codice da uno specifico Web server, delle limitazioni di threading e della gestione delle code perché queste operazioni sono a carico dell'AS.

■ **Estensibilità.** L'architettura modulare degli Application Server e il supporto per i Server e per i moduli applicativi che possono essere caricati dinamicamente, consente alle aziende di estendere facilmente le funzionalità dei loro sistemi e delle relative applicazioni. Di fatto,

l'installazione dei componenti che forniscono queste ulteriori funzionalità permette di migliorare i programmi già in produzione, richiedendo un downtime di sistema minimo. Queste importanti caratteristiche permettono al software dell'AS e alle altre applicazioni un rapido adattamento alle continue mutazioni del mercato.

■ **Robustezza.** L'architettura basata su componenti degli AS e il bilanciamento dinamico dei carichi assicurano l'alta disponibilità dei sistemi. Grazie al supporto delle funzionalità on-line, i componenti del server e la logica applicativa possono essere riconfigurati, aggiunti o rimossi senza interruzioni nell'erogazione dei servizi agli utenti. Queste caratteristiche sono molto importanti per garantire appunto l'alta disponibilità del sistema, requisito necessario per il buon esito delle operazioni mission-critical aziendali.

■ **Sicurezza.** Gli Application Server offrono delle funzioni specifiche di sicurezza end-to-end, necessarie per il processo di esecuzione delle applicazioni aziendali. Per le comunicazioni verso i Web server, vengono impiegati gli algoritmi SSL, HTTPS e le procedure per l'autenticazione HTTP. Per chiudere il potenziale "buco" nella sicurezza rappresentato dalle operazioni transazionali che intercorrono tra i browser e le fonti dei dati, gli AS ricorrono all'autenticazione degli utenti, all'uso dei cookie e ai controlli degli accessi verso i database. Il logging e il tracking degli eventi forniscono una protezione dagli accessi non autorizzati: il log degli eventi infatti consente all'amministratore di individuare i tentativi fatti per scardinare i sistemi di sicurezza e monitorare tutte le altre attività non convenzionali, o comunque non autorizzate.

Uno strumento che non deve assolutamente mancare negli Application Server è un framework che semplifichi la gestione del sistema; per raggiungere questo obiettivo, è necessaria la presenza di tool costruiti in Java. Se si vuole offrire agli amministratori la possibilità di gestire remotamente il sistema, svolgendo il monitoraggio dei server e delle applicazioni tramite un semplice browser, questa scelta risulta praticamente obbligatoria. Le funzioni di amministrazione devono assicurare la gestione di tutto l'ambiente e dei programmi in elaborazione, svolgendo una serie di azioni quali il controllo dell'utilizzo dei server e dei carichi di lavoro, l'adeguamento dell'ambiente per ottenere livelli di prestazione ottimali e il partizionamento dinamico della logica applicativa, ed impiegando svariati Application Server. Lo scopo di questo genere di programmi inclusi negli Application Server è quello di ridurre in modo significativo l'overhead associato alla gestione su larga scala delle applicazioni distribuite.

Grazie agli AS, le applicazioni possono essere sviluppate e distribuite rapidamente. Dopo aver preso in considerazione il discorso relativo allo sviluppo, il passo successivo è quello di affrontare le problematiche legate alla distribuzione ed all'esecuzione, risolte dai moduli di deployment, che semplificano tali attività verso uno o più Application Server.

Il partizionamento delle applicazioni consiste nel distribuire la logica applicativa tra differenti server: a loro volta, i componenti implementati nelle applicazioni su larga scala possono essere raggruppati per facilitarne il partizionamento e la gestione. Un Application Server è quindi un ambiente flessibile, in grado di elaborare la logica applicativa localmente o in modo distribuito: in questo caso, attivando il processo di partizionamento, entrano in gioco tutta una serie di fattori quali la gestione della sicurezza, il livello delle prestazioni, la vicinanza alle risorse di dati, l'ampiezza di banda disponibile sulla rete e la configurazione hardware. La logica applicativa può anche essere riutilizzata e condivisa tra differenti programmi, permettendo così agli sviluppatori di rendere più produttivo il proprio lavoro grazie alla riduzione del tempo necessario alla stesura dei nuovi programmi.

10.4 Uno sguardo al mercato: una confusione voluta?

La confusione a livello di terminologia e la mancanza di una chiara definizione del ruolo e dei confini degli Application Server dipendono in gran parte dalla provenienza dei vari produttori, che si presentano su un mercato con incredibili tassi di crescita. Si tratta di aziende specializzate in strumenti di sviluppo, in database o in strumenti per la gestione delle transazioni?

La definizione sulla quale la maggior parte dei produttori concorda è la seguente: l'Application Server rappresenta il software che funziona sul Middle Tier, ponendosi tra thin o fat client basati su browser Web, database di back-end e pacchetti applicativi. Gli Application Server sono in grado di gestire tutta la logica applicativa e le modalità di connessione implementate nelle applicazioni legacy ed in quelle scritte per gli ambienti client/server. In sostanza, la cosiddetta logica applicativa, responsabile di soddisfare le esigenze procedurali e organizzative delle

imprese, si sposta dal terminale dei mainframe e dai fat-PC dei sistemi client/server verso un più attuale e moderno livello architetturale intermedio, costituito appunto dagli Application Server.

Attualmente, nel turbolento mercato degli Application Server stanno emergendo alcune caratteristiche comuni, riguardanti la convergenza su alcuni standard, facilmente riscontrabili su tutti i prodotti di maggior qualità; tra tutti spicca il supporto alla creazione delle componenti server in modo conforme a COM (Component Object Model), CORBA (Common Object Request Broker Architecture) e EJB (Enterprise JavaBeans). Altri fattori rilevanti sono il supporto al clustering, il bilanciamento dei carichi di lavoro e le funzioni per la gestione dei fail-over. La connessione verso i sistemi legacy quali i mainframe, il supporto per sistemi transazionali e di database nuovi o "datati" rappresentano altre caratteristiche irrinunciabili, così come la capacità di adattamento a qualsiasi tipo di logica applicativa. Da questo punto in avanti cominciano le differenze che connotano ciascun prodotto.

Nella scala dei prezzi "a buon mercato", i produttori di strumenti di sviluppo applicativo considerano gli Application Server come un'estensione dei loro tool e offrono pacchetti combinati di tool di sviluppo e Application Server che consentono di allestire ed avviare sistemi Web abbastanza rapidamente, a un prezzo che generalmente non supera i 100.000 dollari.

All'altro estremo delle categorie di prezzo troviamo i produttori di monitor per il Transaction Processing (sostanzialmente BEA Systems e IBM) che non hanno seguito la direzione presa dagli altri fornitori di AS. I TP Monitor sono infatti utilizzati generalmente dalle applicazioni su rete come gestori di richieste, destinati a trattare le innumerevoli query provenienti dai client, a identificare sistematicamente i database associati alle interrogazioni e ad aggiornarne i dati assicurando la massima garanzia di affidabilità nello svolgimento delle varie operazioni. Questi sistemi di alto livello costituiscono la "classe nobile" degli AS e vengono largamente impiegati anche in attività critiche quali le prenotazioni aeree e la gestione delle carte di credito di molte società. Alcuni produttori, fra i quali IBM con il proprio CICS, sono presenti sul mercato con questo genere di sistemi sin dall'era primordiale dell'informatica. Gli Application Server offrono ai produttori di TP Monitor un nuovo modo per vendere i propri pacchetti agli sviluppatori Web, anche se in quest'area la situazione non è ancora totalmente chiara. Per ora non è facile stabilire chi sarà il vincitore in quanto c'è molta sovrapposizione tra le diverse proposte presenti su un mercato nel quale i fornitori di TP Monitor si orientano ai sistemi più critici per complessità e volumi, per cui non considerano il segmento che punta a realizzare vendite a prezzi bassi.

Una via di mezzo è costituita dai produttori di database e di software per Web server quali Oracle, Sybase, Microsoft, Lotus e via dicendo. Queste aziende sono già in grado di collocare gli AS in prima linea nell'offerta dei propri prodotti. Da un certo punto di vista, Application Server e database vanno infatti di pari passo, per cui i fornitori che operano in quest'ottica puntano ad enfatizzare gli aspetti di connettività di back-end dei loro prodotti, anche a scapito delle capacità di sviluppo. Dal momento che la maggior parte delle grandi aziende hanno ormai adottato il software per la gestione dei database e dei Web server, seguire questo percorso verso gli Application Server per loro potrebbe risultare una facile transizione.

Quindi, mercato ricco mi ci ficco. Sembra essere questa la miglior sintesi di ciò che sta accadendo nel settore degli Application Server.

Benché, apparentemente, il numero dei produttori di software sembri diminuire in seguito ad acquisizioni e processi di consolidamento di dimensioni sempre maggiori, in realtà sul mercato continuano a comparire nuovi attori, rapidissimi nel proporre soluzioni totalmente innovative o frutto della rivisitazione di prodotti già esistenti in chiave più attuale. Così, anche se gli AS costituiscono un segmento di offerta relativamente nuovo e nonostante l'opinione degli analisti preveda che, nel lungo termine, ci sarà spazio per tre o quattro prodotti, attualmente l'offerta è vastissima, ma bisogna andarci molto cauti in quanto non sempre i prodotti presentano un sufficiente grado di maturità e sotto la stessa etichetta vengono inseriti spesso oggetti molto diversi tra loro.

Prova tangibile della ricchezza del mercato è la tabella mostrata nella pagina seguente. Tale tabella è relativa al mercato degli Application Server a Maggio 2000. Vengono indicati il fornitore del prodotto, il nome, la versione più recente e la data di rilascio. Viene inoltre indicato il supporto alle tecnologie EJB, JSP ed XML, e al Java Development Kit. Per ultime vengono indicate le piattaforme supportate.

Fornitore	Prodotto	Versione	Data Rilascio	Supporto				Piattaforme supportate
				EJB	JDK	JSP	XML	
Allaire	Cold Fusion	4.5	gen-99	No	Sì	No	Sì	NT, Solaris, HP-UX
Apple	Web Objects	4.0	gen-99	No	Sì	No	Sì	Mac OS, NT, Solaris, HP-UX
Art Technology Group	Dynamo Application Server	4.05	ott-99	Sì	Sì	Sì		NT, Solaris, IBM-AIX
BEA Systems	Web Logic	4.51	ago-99	No	Sì	Sì		NT, Solaris, HP-UX, IBM-AIX, AS400, Mainframe, Compaq Tru64
Bluestone	Sapphire Web	6.1	giu-99	Sì	Sì	Sì	Sì	NT, Solaris, HP-UX, IBM-AIX, Dec-Alpha, Compaq Tru64
Bullsoft	JOnAS	16.1	gen-00	Sì	Sì	Sì	Sì	NT, Solaris, Linux, IBM-AIX
Chili!Soft	Chili!Soft ASP	1.0	mar-00	No	No	No	No	NT, Solaris, HP-UX, OS/390, Linux
Enhydra	Enhydra	3.0	mar-00	Sì	Sì	Sì	Sì	NT, Linux, UNIX
Exoffice	Intalio Platform	1.0	mar-00	Sì	Sì	Sì	Sì	NT, Solaris, Linux
Evermind	Orion Application Server	1.0	ott-99	Sì	Sì	Sì		NT, Solaris, Linux
Forte	SinexJ Application Server	1.0	set-99	Sì	Sì	Sì	Sì	NT, Solaris
Gemstone	Gemstone/J	3.01	ott-99	Sì	Sì	Sì		NT, Solaris
HAHT Software	HAHT Site	5.0	mar-00	Sì	Sì	Sì	Sì	NT, Solaris, HP-UX, IBM-AIX
IBM	Web Sphere	3.0	set-99	Sì	Sì	Sì	Sì	NT, Solaris, HP-UX, Linux, IBM-AIX, OS/2, OS/390, OS/400, Novell Netware
Inprise	Application Server	4.0	dic-99	Sì	Sì	Sì	Sì	NT, Solaris, HP-UX, Linux, IBM-AIX
Interactive Business Solutions	Enterprise Application Server	1.0	dic-99	Sì	Sì	Sì		NT, Solaris, Linux, IBM-AIX
IONA	iPortal	Beta	ott-99	Sì	Sì	No		NT, Solaris
ObjectSpace, Inc.	Voyager Application Server	3.02	dic-99	Sì	Sì	Sì		NT, Solaris
Oracle	Application Server Oracle 8i	1.0	set-99	Sì	Sì	Sì		NT, Solaris, Linux, Compaq Tru64
Persistence Software	Power Tier 5 for EJB	5.0	set-99	Sì	Sì	Sì		NT, Solaris, HP-UX, IBM-AIX
Pervasive Software	Tango	4.0	nov-99	Sì	Sì	Sì		NT, UNIX
Pranati	Proton EJB	4.02	nov-99	Sì	Sì	Sì		NT, UNIX
Prosyst	Enterprise Beans Server	3.0	dic-99	Sì	Sì	Sì		NT, Solaris, OS/390, OS/400, UNIX, Novell Netware, Linux, HP-UX
Progress Technology	Apptivity	3.2	feb-00	Sì	Sì	Sì	Sì	NT, Solaris, HP-UX, IBM-AIX
Secant	Extreme Enterprise Server	3.5	dic-99	Sì	Sì	Sì		NT, Solaris, HP-UX, Linux
Silver Stream	Application Server	3.0	ott-99	Sì	Sì	Sì		NT, Solaris, HP-UX, IBM-AIX
Sun Microsystems	Net Dynamics	4.0	ago-99	Sì	Sì	Sì		NT, Solaris, HP-UX, Compaq Tru64
Sun Microsystems	iPlanet Application Server	6.0	apr-00	Sì	Sì	Sì		NT, Solaris, HP-UX, IBM-AIX, Compaq Tru64
Sybase	Enterprise Application Server	4.0	gen-00	Sì	Sì	Sì		NT, Solaris
Unify	eWawe Engine	1.1	gen-00	Sì	Sì	Sì		NT, UNIX
Valto	Ejpt	2.0	ott-99	Sì	Sì	Sì		NT, Solaris

Tabella 10.1 Il mercato degli Application Server a Maggio 2000

10.5 Valutazione degli Application Server: guida alla scelta

In origine, si definiva Application Server qualsiasi ambiente in grado di supportare l'esecuzione di applicazioni per i componenti Java lato server. In un secondo tempo, grazie all'evoluzione verso gli Enterprise JavaBeans, gli AS sono divenuti i contenitori per l'esecuzione degli EJB. Partendo da questa premessa, va segnalato che i prodotti attualmente sul mercato che rientrano nella categoria più generale sono oltre 25 e ciascuno è dotato di caratteristiche che lo rendono differente dagli altri. Ognuno di questi pacchetti offre svariate possibilità, ma nessuno di essi è in grado di costituire la scelta universale per ogni situazione: il difficile sta nel capire quale prodotto

soddisfa al meglio i requisiti di ogni situazione specifica. E' quindi necessario pesare attentamente i vantaggi e le capacità dei vari sistemi in relazione alle particolari condizioni nei quali dovranno essere impiegati: ad esempio, alcuni Application Server hanno un grado di adattabilità superiore ad altri, ma questa è una capacità che generalmente costa salata. Altri Application Server offrono invece semplicità e facilità di utilizzo, ma potrebbero non supportare alcune delle funzioni più comuni, come ad esempio la gestione di transazioni eterogenee o le funzioni di fault-tolerance.

Prima di scegliere un determinato Application Server occorre identificare le caratteristiche più rilevanti per la propria azienda, confrontandole successivamente con le caratteristiche presenti negli AS in commercio. In letteratura esistono varie pubblicazioni che illustrano i principali criteri di valutazione da seguire e forniscono le informazioni di background necessarie a comprendere le caratteristiche di un prodotto. Quali sono gli aspetti pratici di cui tenere conto nella scelta e nella gestione di un Application Server?

Le seguenti indicazioni sono state raccolte riassumendo alcuni di questi articoli:

■ **Affidabilità.** Dato che nella gran parte dei casi un Application Server Java è utilizzato per realizzare applicazioni Web orientate alle transazioni, la sua caratteristica più importante è l'affidabilità. L'AS deve, infatti, garantire il funzionamento delle applicazioni in condizioni di carico superiore alla norma e in caso di blocco, ovviamente limitato, di alcune componenti del sistema.

■ **Aderenza agli standard.** Questa è una delle voci più importanti da valutare, poiché da essa dipenderà la capacità del sistema di durare nel tempo seguendo le evoluzioni della tecnologia, eventualmente sostituendo con facilità i componenti non più adeguati alle esigenze del momento. Per tale ragione sono preferibili gli approcci che puntano sugli standard al posto di quelli basati su sistemi proprietari: le API standard accrescono la portabilità, le possibilità di riutilizzo e l'interoperabilità; le API proprietarie diminuiscono, invece, la libertà di scelta. E' consigliabile in particolare l'aderenza agli standard EJB definiti da Sun e agli standard CORBA definiti da OMG, in quanto tali standard sono utilizzati per accedere ai servizi middleware di classe enterprise come le transazioni, il naming e la persistenza. Le API EJB definiscono un modello di componenti Java che supporta la trasportabilità attraverso gli Application Server Java. Un terzo dei programmatori hanno già iniziato a sviluppare applicazioni basate su EJB, non sottovalutando il fatto che il supporto EJB aumenterà di molto l'opportunità di acquistare separatamente applicazioni e componenti. Bisogna tuttavia stare molto attenti agli slogan: benché ciascun produttore affermi la propria, piena adesione agli EJB, molti di essi stanno implementando alcune funzioni che lavorano unicamente sul proprio Application Server, con il rischio quindi di ricadere sulle API proprietarie.

■ **Scalabilità.** La scalabilità, ossia la possibilità di eseguire l'Application Server su più processori e più elaboratori, è un elemento fondamentale per la capacità dell'AS stesso di sopportare carichi crescenti. Meglio essere prudenti già dalle stime iniziali: alcuni analisti consigliano di assicurarsi che l'Application Server prescelto sia in grado di sostenere un carico tre volte superiore a quello previsto sulla carta nelle fasi normali di attività. La maggior parte degli AS Java offrono funzioni di gestione delle risorse che rendono tali sistemi facilmente scalabili. Le caratteristiche minime richieste a un AS consistono nel supporto al session pooling, al multithreading e al database connection pooling. Altri sistemi presentano dei servizi sofisticati quali il bilanciamento dei carichi di lavoro e la replica degli oggetti, oppure possono gestire dinamicamente le risorse di sistema portando temporaneamente gli oggetti non attivi all'esterno dalla memoria di lavoro. Nella scelta dell'Application Server, le domande alle quali bisogna rispondere sono le seguenti: Quanti oggetti al secondo può instanziare il server? Quante transazioni possono essere processate al secondo? Gli oggetti possono essere replicati e distribuiti attraverso server multipli? Quali sono i sistemi di duplicazione e bilanciamento dei carichi di lavoro disponibili?

■ **Fault Tolerance.** Alcuni Application Server implementano il supporto per il recupero automatico in caso di malfunzionamento del sistema. In questo caso bisogna chiedersi: Quali opzioni di ripristino sono disponibili? L'Application Server regge le repliche full state per supportare il fail-over trasparente degli oggetti? Esiste un singolo punto di vulnerabilità?

■ **Indipendenza dagli strumenti e dalle piattaforme.** Come già anticipato, l'indipendenza da implementazioni proprietarie non è sempre totale e assoluta: quindi è necessario valutare diverse possibilità. In quale percentuale il sistema usa API proprietarie? E' richiesto un processo di sviluppo vincolato a strumenti specifici? A quale livello sono collegate le applicazioni specifiche dell'Application Server? E' garantito il supporto alle applicazioni per componenti standard di terze

parti come, ad esempio, gli Enterprise JavaBeans? E' possibile gestire piattaforme multiple, differenti database, protocolli e client diversi?

■ **Facilità di utilizzo e gestione.** Quali sono gli strumenti, forniti con l'Application Server, in grado di semplificare lo sviluppo e la gestione dell'ambiente? E' possibile adattare il nuovo ambiente con le utility di gestione già esistenti?

■ **Accesso ai dati e persistenza.** Quali sono i servizi di accesso ai dati forniti con l'ambiente? E' supportato il data caching all'interno dell'Application Server, necessario per rendere più efficiente l'impiego di uno o più database? Sono supportati i servizi automatici di persistenza (EJB Entity Bean)? Sono consentiti gli accessi simultanei agli oggetti dell'applicazione?

■ **Transazioni.** Un Application Server può supportare le transazioni fornendo direttamente i servizi necessari, oppure dipendendo da un servizio esterno (DBMS). Oltre a questo, bisogna verificare il supporto per i controlli transazionali di aggiornamento sia verso le risorse, sia verso i tradizionali database, che può avvenire tramite file, oggetti o code. Sono gestite le transazioni a semantica lunga come le transazioni nidificate?

■ **Sicurezza.** L'Application Server supporta le procedure di autenticazione, autorizzazione agli accessi e codifica dei dati? Il sistema di sicurezza può adattarsi facilmente con quanto stabilito dalle politiche adottate per il sistema di sicurezza aziendale? Può gestire gli standard Internet di sicurezza come PKI, SSL e X.509?

E' prassi ormai consolidata basarsi sugli EJB e sulla tecnologia per componenti per produrre velocemente le nuove applicazioni: gli EJB semplificano lo sviluppo e consentono la portabilità e il riutilizzo dei componenti applicativi. L'industria degli Application Server, ivi compresi i fornitori di sistemi e i fornitori di applicazioni, appoggia totalmente gli standard EJB e, in particolar modo, i fornitori di pacchetti software sono sicuramente più favorevoli all'implementazione di applicazioni basate su API trasportabili piuttosto che su API esclusive. Di conseguenza, il supporto EJB dovrebbe implicare un'ampia scelta di programmi e di componenti forniti da produttori di terze parti. E' sempre meglio dare la propria preferenza a soluzioni che non richiedano l'impiego di strumenti esclusivi e commercializzati da un solo produttore. Inoltre, a lunga scadenza, la reputazione per la qualità del prodotto offerto risulta molto più importante del time-to-market, specialmente nel mondo delle applicazioni aziendali.

10.6 Verso l'azienda elettronica

Nei prossimi anni, il Web è destinato a sostituire l'architettura client/server. Restano tuttavia da stabilire quali saranno gli standard per i browser ed i server, per i servizi applicativi Web (es. Application Server), per le interfacce dati (XML, EJB, ecc.) e per le interfacce utente in grado di ridurre i costi e aumentare i benefici. Entro il 2003 si prevede un miglioramento dei prodotti e degli standard, portando a un mercato di applicazioni completamente Web-enabled.

Secondo una ricerca del Meta Group, poiché il tempo necessario per sviluppare la prima architettura Web è di circa sei mesi, entro la fine del 2003 il 70% delle società avrà definito un'architettura Web di base. Entro il 2004, le architetture Web saranno parte integrante dei processi di sviluppo e non saranno più considerate entità separate.

La realizzazione di architetture Web dovrebbe soddisfare le seguenti indicazioni:

■ **Principi architetturali del dominio Web.** La strategia Web deve basarsi sulla stessa strategia di mercato adottata dall'azienda: così, l'architettura Web è d'aiuto sia nel raggiungimento degli obiettivi aziendali, sia per consolidare i risultati ottenuti nei confronti dei quadri dirigenziali non informatici.

■ **Documentazione del processo.** Perché sia flessibile e accettato, il processo di sviluppo e modifica dell'architettura Web deve essere specificato, descritto e documentato chiaramente. Le informazioni su come e chi contattare incoraggiano l'interazione tra gli utenti, essenziale nella progettazione delle nuove applicazioni Web.

■ **Web Application Reference Model (WARM).** WARM specifica quali aspetti della tecnologia devono essere considerati anticipatamente dagli utenti. Questo modello mostra come l'utilizzo del Web possa sviluppare vari tipi di interazioni, guidando le decisioni relative allo sviluppo

dell'architettura Web. L'approccio WARM è flessibile, in quanto può essere rivisto e modificato quando cambiano gli assunti di partenza.

■ **Web Application Pattern.** Ogni azienda deve analizzare i pattern che intende usare: alcuni pattern si basano su best-practice di uso comune, ma ogni società dovrà disporre di pattern "su misura" per la propria realtà. Ognuno di questi pattern sarà caratterizzato da una specifica definizione, dalla descrizione del campo d'impiego e da un diagramma architetturale che espliciti i servizi di sistema Web (WSS) che il pattern richiede per il proprio funzionamento.

■ **Workbench.** Una chiara definizione dei ruoli che intervengono nei processi di rilascio delle applicazioni Web (ingegneri e responsabili dei siti Web, creatori di contenuti per l'utente finale) risulta essenziale nella selezione degli strumenti. Come abbiamo visto, i Web Application Pattern definiscono i WSS necessari per la progettazione dei sistemi Web; i Workbench specificano invece gli strumenti (non associati a uno specifico Middleware) consigliati per gli utenti finali (ad esempio, i responsabili di siti Web hanno bisogno di tool per monitorare lo stato e le prestazioni del sito, mentre i creatori di contenuti devono sviluppare sofisticati file grafici e multimediali).

■ **Profilo di rischio.** Bisogna porsi la domanda: qual'è il livello di rischio che l'azienda è disposta ad affrontare per la tecnologia Web? La risposta deve venire dal gruppo di lavoro, che deve descrivere in quale "stadio evolutivo" si trovino le tecnologie adottate e istruire di conseguenza i programmatori.

■ **Link esterni all'architettura.** Per essere competitiva, un'architettura Web deve presentare scopi e limiti ben definiti, proponendosi come modulo attivo dell'intera infrastruttura d'impresa. Questa situazione impone che siano forniti esplicitamente i collegamenti verso quei documenti, esterni all'architettura, che descrivono le interfacce chiave per l'architettura Web e per quella aziendale (architettura di rete, firewall e amministrazione dei database).

Appendice Programmazione Object Oriented e Java

A.1 Introduzione al paradigma Object Oriented

Il modello classico, conosciuto come paradigma procedurale, può essere riassunto in due parole: "Divide et Impera" ossia dividi e conquista. Secondo il **paradigma procedurale**, infatti, un problema complesso viene suddiviso in problemi più semplici in modo che siano facilmente risolvibili mediante programmi procedurali. E' chiaro che in questo caso, l'attenzione del programmatore è accentrata al problema. A differenza del primo, il **paradigma Object Oriented** accentra l'attenzione verso i dati. L'applicazione viene suddivisa in un insieme di oggetti in grado di interagire tra loro e codificati in modo tale che la macchina sia in grado di comprenderli.

Quando i programmi erano scritti in **Assembler**, ogni dato era globale e le funzioni andavano disegnate a basso livello. Con l'avvento dei linguaggi procedurali come il **C**, i programmi sono diventati più robusti e semplici da mantenere dato che il linguaggio fornisce regole sintattiche e semantiche che, supportate da un compilatore, consentono un maggior livello di astrazione rispetto a quello fornito dall'Assembler, e forniscono un ottimo supporto alla decomposizione procedurale dell'applicazione. Con l'aumento delle prestazioni dei calcolatori e di conseguenza con l'aumento della complessità delle applicazioni, l'approccio procedurale ha iniziato a mostrare i propri limiti rendendo necessario definire un nuovo modello e nuovi linguaggi di programmazione. I linguaggi come **Java** e **C++** sono il supporto ideale al disegno ad oggetti di applicazioni fornendo un insieme di regole sintattiche e semantiche che aiutano nello sviluppo di oggetti. Il paradigma Object Oriented formalizza la tecnica di incapsulare e raggruppare parti di un programma. In generale, il programmatore divide le applicazioni in gruppi logici che rappresentano concetti sia a livello di utente sia a livello applicativo. I pezzi vengono poi riuniti a formare un'applicazione. Scendendo nei dettagli, lo sviluppatore ora inizia con l'analizzare tutti i singoli aspetti concettuali che compongono un programma. Questi concetti sono chiamati **oggetti** ed hanno nomi legati a ciò che rappresentano. Una volta che gli oggetti sono identificati, il programmatore decide di quali attributi (**dati**) e funzionalità (**metodi**) dotare le entità. L'analisi infine dovrà includere le modalità di interazione tra gli oggetti. Proprio grazie a queste interazioni sarà possibile riunire gli oggetti a formare un'applicazione.

A differenza di procedure e funzioni, gli oggetti sono **auto-documentanti**, cioè rappresentano entità ben definite. Un'applicazione può essere scritta a partire da poche informazioni, ed in particolar modo il funzionamento interno delle funzionalità di ogni oggetto è completamente nascosto al programmatore (incapsulamento).

L'OOP rappresenta il cuore di Java. Tutti i linguaggi di programmazione orientati agli oggetti offrono meccanismi che aiutano ad implementare il modello orientato agli oggetti: tali meccanismi sono l'incapsulamento, l'ereditarietà ed il polimorfismo.

A.1.1 Ereditarietà

L'**ereditarietà** è il processo tramite il quale un oggetto acquisisce le proprietà di un altro. Questo è importante perché sostiene il concetto della classificazione gerarchica. Utilizzando l'ereditarietà, un oggetto può limitarsi a definire le qualità che lo rendono unico all'interno di una classe. Può ereditare gli attributi generali dai genitori. Quindi, è il meccanismo dell'ereditarietà che consente ad un oggetto di essere un'istanza specifica di una classe più generale (concetti di superclasse e sottoclasse). L'ereditarietà interagisce anche con l'incapsulamento. Se una data classe incapsula alcuni attributi, allora qualsiasi sottoclasse comprenderà gli stessi attributi più tutti quelli aggiuntivi dipendenti dalla sua specializzazione.

L'organizzazione degli oggetti fornita dal meccanismo di ereditarietà semplifica le operazioni di **manutenzione** di un'applicazione. Ogni volta che si rende necessaria una modifica, è in genere sufficiente creare un nuovo oggetto all'interno di una classe di oggetti ed utilizzare tale oggetto per rimpiazzarne uno vecchio ed obsoleto.

Un altro vantaggio dell'ereditarietà è la **riusabilità del codice**. Creare una classe di oggetti per definire entità è molto di più che crearne una semplice rappresentazione: per la maggior parte delle classi l'implementazione è spesso scritta all'interno della descrizione. In Java ad esempio ogni volta che definiamo un concetto, esso viene definito come una classe, all'interno della quale

è scritto il codice necessario ad implementare le funzionalità dell'oggetto per quanto generico esso sia. Se viene creato un nuovo oggetto (e quindi una nuova classe) a partire da un oggetto (classe) esistente si dice che la nuova classe deriva dalla originale. Quando questo accade, tutte le caratteristiche dell'oggetto principale diventano parte della nuova classe. Dal momento che la classe derivata eredita le funzionalità della classe predecessore, l'ammontare del codice da scrivere necessario per la nuova classe è pesantemente ridotto: il codice della classe di origine è stato riutilizzato.

A questo punto è necessario iniziare a definire formalmente alcuni termini. La relazione di ereditarietà tra classi è espressa in termini di superclasse e sottoclasse. Una **superclasse** è la classe più generica utilizzata come punto di partenza per derivare nuove classi. Una **sottoclasse** rappresenta, invece, una specializzazione di una superclasse. E' uso comune chiamare una superclasse classe base e una sottoclasse classe derivata. Questi termini sono comunque relativi in quanto una classe derivata può a sua volta essere una classe base per una classe più specifica.

A.1.2 Incapsulamento

L'**incapsulamento** è il meccanismo che lega il codice e i dati che questo manipola e assicura entrambi dalle interferenze esterne e dagli abusi. Un sistema per pensare all'incapsulamento consiste nell'immaginarlo come un involucro protettivo che impedisce un accesso arbitrario al codice e ai dati da parte di un altro codice definito all'esterno dell'involucro. L'accesso al codice e ai dati all'interno dell'involucro è rigidamente controllato da un'interfaccia ben definita.

In Java il concetto base dell'incapsulamento è la classe. Una **classe** definisce la struttura ed il comportamento (dati e codice) che verranno condivisi da una serie di oggetti. Al momento della creazione di una classe, vengono specificati i membri che la costituiscono, cioè le variabili di istanza (dati) e i metodi (codice che opera sui dati). Dato che lo scopo di una classe è incapsulare la complessità, esistono dei meccanismi per nascondere la complessità dell'implementazione all'interno della classe. Ogni metodo o variabile di una classe può essere contrassegnato come pubblico o privato (può accedervi solo un codice che sia membro della classe). Dato che ai membri privati di una classe possono accedere solo altre parti del programma mediante i metodi pubblici della classe, tale meccanismo garantisce che non si verificheranno azioni improprie. Ovviamente, questo significa che l'interfaccia pubblica dovrà essere progettata con attenzione, in modo da non esporre una quantità eccessiva del funzionamento interno della classe.

Come già discusso, nella programmazione orientata ad oggetti si definiscono oggetti creando rappresentazioni di entità o nozioni da utilizzare come parte di un'applicazione. Per assicurarci che il programma lavori correttamente ogni oggetto deve rappresentare in modo corretto il concetto di cui è modello senza che l'utente possa disgregarne l'integrità.

Per fare questo è importante che l'oggetto esponga solo la porzione di codice e dati che il programma deve utilizzare. Ogni altro dato e codice deve essere nascosto affinché sia possibile mantenere l'oggetto in uno stato consistente. Ad esempio, se un oggetto rappresenta uno stack di dati, l'applicazione dovrà poter accedere solo al primo dato dello stack, ossia alle funzioni di Push e Pop. Il contenitore ed ogni altra funzionalità necessaria alla sua gestione dovrà essere protetta rispetto all'applicazione, garantendo così che l'unico errore in cui si può incorrere è quello di inserire un oggetto sbagliato in testa allo stack o estrapolare più dati del necessario. In qualunque caso l'applicazione non sarà mai in grado di creare inconsistenze nello stato del contenitore. L'incapsulamento inoltre localizza tutti i possibili problemi in porzioni ristrette di codice. Un'applicazione potrebbe inserire dati sbagliati nello stack, ma saremo comunque sicuri che l'errore è localizzato all'esterno dell'oggetto.

Una volta che un oggetto è stato incapsulato e testato, tutto il codice ed i dati associati sono protetti. Modifiche successive al programma non potranno causare rotture nelle dipendenze tra gli oggetti dato che non saranno in grado di vedere i legami tra dati ed entità. L'effetto principale sull'applicazione sarà quindi quello di localizzare i bug evitando la propagazione di errori, dotando così l'applicazione di grande stabilità.

In un programma decomposto per funzioni, le procedure tendono ad essere interdipendenti. Ogni modifica al programma richiede spesso la modifica di funzioni condivise, cosa che può propagare un errore alle componenti del programma che le utilizzano. In un programma Object Oriented, le dipendenze sono sempre strettamente sotto controllo e sono mascherate all'interno delle entità concettuali. Modifiche a programmi di questo tipo riguardano tipicamente l'aggiunta di nuovi oggetti, ed il meccanismo di ereditarietà ha l'effetto di preservare l'integrità referenziale

delle entità componenti l'applicazione. Se invece fosse necessario modificare internamente un oggetto, le modifiche sarebbero comunque limitate al corpo dell'entità e quindi confinate all'interno dell'oggetto che impedirà la propagazione di errori all'esterno del codice. Anche le operazioni di ottimizzazione risultano semplificate. Quando un oggetto risulta avere performance molto basse, si può cambiare facilmente la sua struttura interna senza dovere riscrivere il resto del programma, purché le modifiche non tocchino le proprietà già definite dell'oggetto.

A.1.3 Polimorfismo

Il **polimorfismo** (termine derivante dal greco, che significa "molte forme") è una caratteristica che consente a un'interfaccia di essere utilizzata per una classe generale di azioni. L'azione specifica viene determinata dall'esatta natura della situazione.

In termini più generali, il concetto di polimorfismo viene espresso dalla frase "**un'interfaccia, più metodi**", a significare che è possibile disegnare un'interfaccia generica per un gruppo di attività correlate. Questo contribuisce a ridurre la complessità permettendo di utilizzare la stessa interfaccia per specificare una classe generale di azioni.

A.2 Introduzione a Java

Java fu concepito da James Gosling, Patrick Naughton, Chris Warth, Ed Frank e Mike Sheridan presso Sun Microsystems, nel 1991. Furono necessari 18 mesi per sviluppare la prima versione funzionante. All'inizio, questo linguaggio venne battezzato **Oak**, ma nel 1995 il nome venne modificato in Java. Fatto piuttosto sorprendente, la spinta originale non provenne da Internet. Infatti, la motivazione principale era la necessità di un linguaggio indipendente dalla piattaforma che potesse essere utilizzato per creare software da incorporare in vari dispositivi di elettronica di consumo, come i forni a microonde e i telecomandi.

Il problema del C e del C++ (e della maggior parte degli altri linguaggi) è che sono concepiti per essere compilati per una destinazione specifica. Sebbene sia possibile compilare un programma C++ per qualsiasi tipo di CPU, per riuscirci è necessario un intero compilatore C++ mirato per quella specifica CPU. Ma i compilatori sono costosi e la loro creazione richiede molto tempo. Era necessaria una soluzione più semplice e più vantaggiosa dal punto di vista economico. Gosling & Co. incominciarono a lavorare su un linguaggio portatile e sicuro, indipendente dalla piattaforma, che potesse essere utilizzato per produrre un codice che girasse su una vasta gamma di CPU sotto diversi ambienti. Alla fine questo sforzo condusse alla creazione di Java.

A.2.1 La magia di Java: i bytecode

La chiave che consente a Java di risolvere sia i problemi di sicurezza, sia quelli di portabilità è il fatto che l'output di un compilatore Java non è un codice eseguibile. Invece, si tratta di un **bytecode**, un insieme altamente ottimizzato di istruzioni concepite per essere eseguite da una macchina virtuale emulata dal sistema runtime di Java (la **Java Virtual Machine**): la JVM traduce le istruzioni dei codici binari indipendenti dalla piattaforma generati dal compilatore Java, in istruzioni eseguibili dalla macchina locale. In altre parole, il sistema runtime è un interprete del bytecode.

Dato che i programmi Java vengono interpretati piuttosto che compilati, come avviene per esempio per programmi C++ compilati su un codice eseguibile, è molto più semplice eseguirli in una vasta gamma di ambienti. La ragione è evidente: è sufficiente implementare il sistema runtime di Java su ogni piattaforma.

Il fatto che Java venga interpretato contribuisce anche a renderlo sicuro. Dato che l'esecuzione di ogni programma Java avviene sotto il controllo del sistema runtime, questo può contenere il programma e impedire che generi effetti indesiderati all'esterno del sistema.

La natura di linguaggio a oggetti di Java consente di sviluppare applicazioni utilizzando oggetti concettuali piuttosto che procedure e funzioni. La sintassi object oriented di Java supporta la creazione di oggetti concettuali, il che consente al programmatore di scrivere codice stabile e riutilizzabile, utilizzando il paradigma object oriented secondo il quale appunto il programma è scomposto in concetti piuttosto che funzioni o procedure.

La sua stretta parentela con il linguaggio C a livello sintattico fa sì che un programmatore che abbia già fatto esperienza con linguaggi come C, C++, Perl sia facilitato nell'apprendimento del linguaggio.

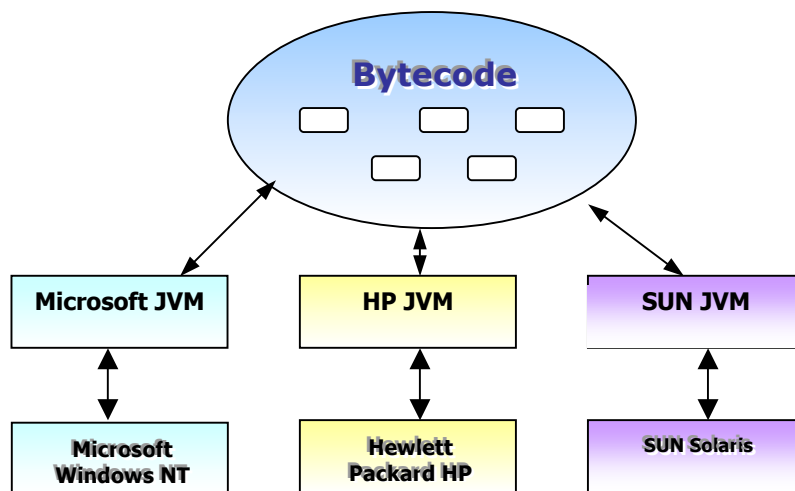


Figura A.1 Architettura di un'applicazione Java

A.2.2 Le caratteristiche essenziali di Java

Di seguito un elenco delle caratteristiche essenziali di Java:

- Semplice, perché eredita la sintassi del C/C++
- Sicuro
- Portabile
- Orientato agli oggetti
- Solido
- Multithreaded
- Indipendente dall'architettura (scrivere una volta, eseguire ovunque, in qualsiasi momento)
- Interpretato
- Con prestazioni elevate
- Distribuito
- Dinamico

A.2.2.1 Indipendenza dalla piattaforma

Le istruzioni binarie di Java indipendenti dalla piattaforma sono più comunemente conosciute come bytecode. Il bytecode di Java è prodotto dal compilatore e ha bisogno di uno strato di software, la Java Virtual Machine, per essere eseguito. La JVM è un programma scritto mediante un qualunque linguaggio di programmazione dipendente dalla piattaforma, e traduce le istruzioni Java, nella forma di bytecode, in istruzioni native del processore locale.

Non essendo il bytecode legato ad una particolare architettura hardware, questo fa sì che per trasferire un'applicazione Java da una piattaforma ad un'altra è necessario solamente che la nuova piattaforma sia dotata di un'apposita JVM. In presenza di un interprete, un'applicazione Java potrà essere eseguita su qualunque piattaforma senza necessità di essere ricompilata.

In alternativa, si possono utilizzare strumenti come i **Just In Time Compiler** (JIT), compilatori in grado di tradurre il bytecode in un formato eseguibile su una specifica piattaforma al momento dell'esecuzione del programma Java. I vantaggi nell'uso dei compilatori JIT sono molteplici. La tecnologia JIT traduce il bytecode in un formato eseguibile al momento del caricamento della applicazione. Ciò consente di migliorare le performance dell'applicazione che non dovrà più passare per la JVM, e allo stesso tempo preserva la caratteristica di portabilità del codice. L'unico svantaggio nell'uso di un compilatore JIT sta nella perdita di prestazioni al momento del lancio della applicazione, che deve essere prima compilata e poi eseguita.

Un ultimo aspetto interessante della tecnologia Java è quello legato agli sviluppi che la tecnologia sta avendo. Negli ultimi anni molti produttori di hardware hanno iniziato a rilasciare processori in grado di eseguire direttamente il bytecode di Java a livello di istruzioni macchina senza l'uso di una virtual machine.

A.2.2.2 Uso della memoria e multithreading

Un problema scottante quando si parla di programmazione è la gestione e l'uso della memoria. Uno dei problemi più complessi da affrontare quando si progetta un'applicazione di fatto è proprio quello legato al mantenimento degli spazi di indirizzamento del programma, con il risultato che spesso è necessario sviluppare complesse routine specializzate nella gestione e tracciamento della memoria assegnata all'applicazione.

Java risolve il problema alla radice sollevando direttamente il programmatore dall'onere della gestione della memoria grazie ad un meccanismo detto **garbage collection**. Il garbage collector tiene traccia degli oggetti utilizzati da una applicazione Java, nonché delle referenze a tali oggetti. Ogni volta che un oggetto non è più referenziato per tutta la durata di una specifica slide di tempo, viene rimosso dalla memoria e la risorsa liberata viene di nuovo messa a disposizione dell'applicazione che potrà continuare a farne uso. Questo meccanismo è in grado di funzionare correttamente in quasi tutti i casi anche se molto complessi, ma non si può affermare che è completamente esente da problemi. Esistono, infatti, dei casi documentati di fronte ai quali il garbage collector non è in grado di intervenire. Un caso tipico è quello della **referenza circolare**, in cui un oggetto A referencia un oggetto B e viceversa, ma l'applicazione non sta utilizzando nessuno dei due.

Java è un linguaggio multithreaded. Il **multithreading** consente ad applicazioni Java di sfruttare il meccanismo di concorrenza logica. Parti separate di un programma possono essere eseguite come se fossero (dal punto di vista del programmatore) processate parallelamente. L'uso di thread rappresenta un modo semplice di gestire la concorrenza tra processi in quanto gli spazi di indirizzamento della memoria dell'applicazione sono condivisi con i thread eliminando così la necessità di sviluppare complesse procedure di comunicazione tra processi.

Altra interessante caratteristica di Java è che supporta il metodo detto di **Dynamic Loading and Linking**. Secondo questo metodo, ogni modulo del programma (classe) è memorizzato in un determinato file. Quando un programma Java viene eseguito, le classi vengono caricate e istanziate solo al momento del loro effettivo utilizzo. Una applicazione composta da molte classi caricherà solamente quelle porzioni di codice che devono essere eseguite in un determinato istante. Infine Java prevede un gran numero di classi precompilate che forniscono funzionalità come strumenti di I/O o di networking.

A.2.3 Tipi primitivi, strutture sintattiche e controllo di flusso

La tabella proposta di seguito descrive brevemente i tipi primitivi disponibili con Java.

Tipo primitivo	Bit	Descrizione
boolean	1	può assumere esclusivamente i due valori true e false, l'assegnazione di un valore può avvenire in modo classico o attraverso una condizione (es: data la variabile b di tipo boolean il valore false gli può essere assegnato attraverso l'istruzione b=false oppure b=10>1)
byte	8	può memorizzare valori interi compresi tra -128 e 127
short	16	può memorizzare valori interi compresi tra -32768 e 32767
int	32	può memorizzare valori interi compresi tra -2147483648 e 2147483647
long int	64	può memorizzare valori interi compresi tra -9223372036854775808 e 9223372036854775807
float	32	tipo a virgola mobile, può memorizzare valori sia positivi che negativi (anche decimali) compresi tra $-3,4 \cdot 10^{38}$ e $3,4 \cdot 10^{38}$
double	64	tipo a virgola mobile, può memorizzare valori sia positivi che negativi (anche decimali) compresi tra $-1,7 \cdot 10^{308}$ e $1,7 \cdot 10^{308}$
char	16	può memorizzare valori positivi interi compresi tra 0 e 65535, di solito utilizzati per la memorizzazione di caratteri ASCII

Tabella A.1 Tipi primitivi in Java

ARRAY MONODIMENSIONALI tipo nomeArray[];

ARRAY MULTIDIMENSIONALI tipo nomeArray[][];

ISTRUZIONE IF-ELSE

```
if (cond) {
    istruzione 1;
    istruzione 2;
    ...
    istruzione n;
}
else {
    istruzione n+1;
    istruzione n+2;
    ...
    istruzione m;
} //else
```

ISTRUZIONE SWITCH

```
switch (espr){
    case valore1:
        istruzione 1;
        ...
        [break;]
    case valore2:
        istruzione 2;
        ...
        [break;]
    ...
    case valore n:
        istruzione n;
        ...
        [break;]
    default:
        istruzione default;
        ...
        [break;]
} //switch
```

ISTRUZIONE WHILE

```
while (cond) {
    istruzione 1;
    istruzione 2;
    ...
    istruzione n;
} //while
```

ISTRUZIONE FOR

```
for (espressione di assegnazione; condizione; espressione di incremento){
    istruzione 1;
    istruzione 2;
    ...
    istruzione n;
} //for
```

ISTRUZIONE DO-WHILE

```
do{
    istruzione 1;
    istruzione 2;
    ...
    istruzione n;
} while (cond);
```


A.2.4 Metodi

I metodi sono gruppi di istruzioni riuniti a fornire una singola funzionalità. Hanno una sintassi molto simile a quella della definizione di funzioni ANSI C, e possono essere rappresentati con la seguente forma:

```
return_type method_name (arg_type name [,arg_type name] )
{
    istruzioni
}
```

dove `return_type` e `arg_type` rappresentano un qualsiasi tipo di dato (primitivo o oggetto), `name` e `method_name` sono identificatori alfanumerici ed iniziano con una lettera. Se il metodo non ritorna valori, dovrà essere utilizzata la chiave speciale **void** al posto di `return_type`. Se il corpo di un metodo contiene dichiarazioni di variabili, queste saranno visibili solo all'interno del metodo stesso, ed il loro ciclo di vita sarà limitato all'esecuzione del metodo. Non manterranno il loro valore tra chiamate differenti e, non saranno accessibili da altri metodi.

Per evitare l'allocazione di variabili inutilizzate, il compilatore Java prevede una forma di controllo secondo la quale un metodo non può essere compilato se esistono variabili cui non è stato assegnato alcun valore. In questi casi la compilazione verrà interrotta con la restituzione di un messaggio di errore.

A.2.5 Classi

Una classe Java deve rappresentare un oggetto concettuale. Per poterlo fare deve raggruppare dati e metodi assegnando un nome comune. La sintassi è la seguente:

```
class ObjectName
{
    data_declarations
    method_declarations
}
```

I dati ed i metodi contenuti all'interno della classe vengono chiamati **membri** della classe. E' importante notare che dati e metodi devono essere rigorosamente definiti all'interno della classe. Non è possibile, in nessun modo, dichiarare variabili globali, funzioni o procedure. Questa restrizione del linguaggio Java scoraggia il programmatore ad effettuare una decomposizione procedurale, incoraggiando di conseguenza ad utilizzare l'approccio object oriented.

Java fa una netta distinzione tra classi e tipi primitivi. Una delle maggiori differenze è che un oggetto non è allocato dal linguaggio al momento della dichiarazione.

A.2.6 Package Java

I **package** sono meccanismi per raggruppare definizioni di classe in librerie, allo stesso modo di altri linguaggi di programmazione. Il meccanismo è provvisto di una struttura gerarchica per l'assegnamento di nomi alle classi in modo da evitare eventuali collisioni nel caso in cui alcuni programmatori usino lo stesso nome per differenti definizioni di classe.

Oltre a questo, sono molti i benefici nell'uso di questo meccanismo. Primo, le classi possono essere mascherate all'interno dei package implementando l'incapsulamento anche a livello di file. Secondo, le classi di un package possono condividere dati e metodi con classi di altri package. Terzo, i package forniscono un meccanismo efficace per distribuire oggetti.

A.2.7 Il modificatore public

Di default, la definizione di una classe Java può essere utilizzata solo dalle classi all'interno del suo stesso package. Java richiede al programmatore di esplicitare quali classi e quali membri possano essere utilizzati all'esterno del package. A questo scopo riserva il modificatore **public** da utilizzare prima della dichiarazione della classe o di un membro. Le specifiche del linguaggio richiedono che il codice sorgente di classe pubblica sia memorizzata in un file avente lo stesso nome della classe (includere maiuscole e minuscole), ma con estensione **.java**. Come conseguenza

alla regola, può esistere solo una classe pubblica per ogni file di sorgente. Questa regola è rafforzata dal compilatore che scrive il bytecode di ogni classe in un file avente lo stesso nome della classe (incluse maiuscole e minuscole), ma con estensione **.class**.

Lo scopo di questa regola è di semplificare la ricerca di sorgenti e bytecode da parte del programmatore. Per esempio, si supponga di avere tre classi A, B e C in un file unico. Se A fosse la classe pubblica (e solo una lo può essere), il codice sorgente di tutte e tre le classi dovrebbe trovarsi all'interno di un file A.java. Quando A.java verrà compilato, il compilatore creerà una classe per ogni classe nel file: A.class, B.class e C.class.

Questa organizzazione, per quanto contorta, ha un senso logico. Se, come detto, la classe pubblica è l'unica a poter essere eseguita da altre classi all'esterno del package, le altre classi rappresentano solo l'implementazione di dettagli non necessari al di fuori del package.

A.2.8 L'istruzione import

Il runtime di Java fornisce un ambiente completamente dinamico. Le classi non sono caricate fino a che non sono referenziate per la prima volta durante l'esecuzione dell'applicazione. Questo consente di ricompilare singole classi senza dover ricaricare grandi applicazioni.

Poiché ogni classe Java è memorizzata in un suo file, la JVM può trovare i file binari .class appropriati cercando nelle directory specificate nella variabile d'ambiente **CLASSPATH**. Inoltre, dal momento che le classi possono essere organizzate in package, è necessario specificare a quale package una classe appartenga, pena l'incapacità della JVM di trovarla.

Un modo per indicare il package cui una classe appartiene è quello di specificarlo ad ogni chiamata alla classe ossia utilizzando **nomi qualificati** (tecnicamente, in Java un nome qualificato è un nome formato da una serie di identificatori separati da punto per identificare univocamente una classe). L'uso di nomi qualificati non è sempre comodo, soprattutto per package organizzati con gerarchie a molti livelli. Per venire incontro al programmatore, Java consente di specificare una volta per tutte il nome qualificato di una classe all'inizio del file utilizzando la parola chiave **import**. L'istruzione import ha come unico effetto quello di identificare univocamente una classe e quindi di consentire al compilatore di risolvere nomi di classe senza ricorrere ogni volta a nomi qualificati.

A.3 Incapsulamento in Java

L'incapsulamento di oggetti è il processo di mascheramento dei dettagli dell'implementazione ad altri oggetti per evitare riferimenti incrociati. I programmi scritti con questa tecnica risultano molto più leggibili e limitano i danni dovuti alla propagazione di un bug.

Nascondendo i dettagli, possiamo assicurare a chi utilizza l'oggetto che ciò che sta utilizzando è sempre in uno stato consistente a meno di bug dell'oggetto stesso. Uno stato consistente è uno stato permesso dal disegno di un oggetto. E' però importante notare che uno stato consistente non corrisponde sempre allo stato aspettato dall'utente dell'oggetto. Se infatti l'utente trasmette all'oggetto parametri errati, l'oggetto si troverà in uno stato consistente, ma non nello stato desiderato.

A.3.1 Modificatori public, private e protected

Java fornisce supporto per l'incapsulamento a livello di linguaggio mediante i modificatori **public** e **private** da utilizzare al momento della dichiarazione di variabili e metodi. I membri di una classe, o l'intera classe, definiti **public**, sono liberamente accessibili da ogni classe utilizzata all'interno dell'applicazione. I membri di una classe definiti **private** possono essere utilizzati solo dai membri della stessa classe. I membri privati mascherano i dettagli dell'implementazione di una classe.

Membri di una classe non dichiarati né **public**, né **private** saranno per definizione accessibili a tutte le classi dello stesso package. Questi membri o classi sono comunemente detti **package friendly**.

Il modificatore **private** realizza incapsulamento a livello di definizione di classe e serve a definire membri che devono essere utilizzati solo da altri membri della stessa classe di definizione. L'intento, di fatto, è di nascondere porzioni di codice della classe che non devono essere utilizzati da altre classi.

Il modificatore **public** consente di definire classi o membri di una classe visibili da qualsiasi classe all'interno dello stesso package e non. Public deve essere utilizzato per definire l'interfaccia che l'oggetto mette a disposizione dell'utente. Tipicamente metodi membro pubblici utilizzano membri privati per implementare le funzionalità dell'oggetto.

Un altro modificatore messo a disposizione dal linguaggio Java è **protected**. Membri di una classe dichiarati protected possono essere utilizzati sia dai membri della stessa classe che da altre classi purché appartenenti allo stesso package.

A.3.2 L'operatore new

Per creare un oggetto attivo sin dalla sua definizione di classe, Java mette a disposizione l'operatore **new**. Questo operatore è paragonabile alla **malloc** in C, ed è identico allo stesso operatore in C++. Oltre a generare un oggetto, new consente di assegnargli lo stato iniziale ritornando un riferimento (indirizzo di memoria) al nuovo oggetto che può essere memorizzato in una **variabile reference** di tipo compatibile mediante l'operatore di assegnamento =.

La responsabilità della gestione della liberazione della memoria allocata per l'oggetto non più in uso è del garbage collector. Per questo motivo, a differenza di C++, Java non prevede alcun meccanismo esplicito per distruggere un oggetto creato.

La sintassi dell'operatore new prevede un tipo seguito da un insieme di parentesi. Le parentesi indicano che per creare l'oggetto verrà chiamata un metodo chiamato costruttore, responsabile dell'inizializzazione dello stato dell'oggetto.

A.3.3 Costruttori

Tutti i programmatori, esperti e non, conoscono il pericolo che costituisce una variabile non inizializzata. Ad esempio, fare un calcolo matematico con una variabile intera non inizializzata può generare risultati errati. In un'applicazione object oriented, un oggetto è un'entità molto più complessa di un tipo primitivo come int e, l'errata inizializzazione dello stato dell'oggetto può essere causa della terminazione prematura dell'applicazione o della generazione di bug (errori) intermittenti difficilmente controllabili. In molti altri linguaggi di programmazione, il responsabile dell'inizializzazione delle variabili è il programmatore. In Java questo è impossibile dal momento che potrebbero esistere dati membro privati dell'oggetto e quindi inaccessibili all'oggetto utente.

I **costruttori** sono metodi speciali chiamati quando viene creata una nuova istanza di classe e servono ad inizializzare lo stato iniziale dell'oggetto. Questi metodi hanno lo stesso nome della classe di cui sono membro e non restituiscono nessun tipo. Se una classe non è provvista di costruttore, Java ne utilizza uno speciale di default, che non fa nulla. Dal momento che il linguaggio garantisce la chiamata al costruttore ad ogni istanziamento di un oggetto, un costruttore scritto intelligentemente garantisce che tutti i dati membro vengano inizializzati.

Al programmatore è consentito scrivere più di un costruttore per una data classe a seconda delle necessità di disegno dell'oggetto (overloading o sovraccarico dei metodi), permettendogli di passare diversi insiemi di dati di inizializzazione.

Java permette una sola chiamata a costruttore al momento della referenziazione dell'oggetto, ponendo quindi una importante restrizione sulla chiamata. Questo significa che nessun costruttore può essere eseguito nuovamente dopo la creazione dell'oggetto.

Java consente ad un costruttore di chiamare altri costruttori appartenenti alla stessa definizione di classe (**cross calling** tra costruttori). Questo meccanismo è utile dal momento che i costruttori generalmente hanno funzionalità simili e, un costruttore che assume uno stato di default, potrebbe chiamarne uno che prevede che lo stato sia passato come parametro, chiamandolo e passando i dati di default. Per chiamare un costruttore da un altro, è necessario utilizzare una sintassi speciale:

```
this(parameter_list);
```

Nella chiamata, parameter_list rappresenta la lista di parametri del costruttore che si intende chiamare. Una chiamata cross-call tra costruttori, deve essere la prima riga di codice del costruttore chiamante. Qualsiasi altra cosa venga fatta prima, compresa la definizione di variabili, Java non consente di effettuare tale chiamata. Il costruttore corretto viene determinato in base alla lista dei parametri paragonando parameter_list con la lista dei parametri di tutti i costruttori della classe.

A.4 Ereditarietà in Java

L'ereditarietà è la caratteristica dei linguaggi object oriented che consente di utilizzare classi come base per la definizione di nuovi oggetti che ne specializzano il concetto. L'ereditarietà fornisce inoltre un ottimo meccanismo per aggiungere funzionalità ad un programma con rischi minimi per le funzionalità esistenti, nonché un modello concettuale che rende un programma object oriented auto-documentante rispetto ad un analogo scritto con linguaggi procedurali. Per utilizzare correttamente l'ereditarietà, il programmatore deve conoscere a fondo gli strumenti forniti dal linguaggio in supporto.

A.4.1 Disegnare una classe base

Quando disegniamo una classe, dobbiamo sempre tenere a mente che con molta probabilità ci sarà qualcuno che in seguito potrebbe aver bisogno di utilizzarla tramite il meccanismo di ereditarietà. Ogni volta che si utilizza una classe per ereditarietà ci si riferisce a questa come alla classe base o **superclasse**. Il termine ha come significato che la classe è stata utilizzata come fondamenta per una nuova definizione. Quando definiamo nuovi oggetti utilizzando l'ereditarietà, tutte le funzionalità della classe base sono trasferite alla nuova classe detta classe derivata o **sottoclasse**. Quando si fa uso dell'ereditarietà, bisogna sempre tener ben presente alcuni concetti. L'ereditarietà consente di utilizzare una classe come punto di partenza per la scrittura di nuove classi. Questa caratteristica può essere vista come una forma di riutilizzo del codice: i membri della classe base sono "concettualmente" copiati nella nuova classe. Come conseguenza diretta, l'ereditarietà consente alla classe base di modificare la superclasse. In altre parole, ogni aggiunta o modifica ai metodi della superclasse sarà applicata solo alla classe derivata. La classe base risulterà quindi protetta dalla generazione di nuovi eventuali bug, che rimarranno circoscritti alla classe derivata. La classe derivata per ereditarietà supporterà tutte le caratteristiche della classe base.

In definitiva, tramite questa tecnica è possibile creare nuove varietà di entità già definite mantenendone tutte le caratteristiche e le funzionalità. Questo significa che se un'applicazione è in grado di utilizzare una classe base, sarà in grado di utilizzarne la derivata allo stesso modo. Per questi motivi, è importante che una classe base rappresenti le funzionalità generiche delle varie specializzazioni che andremo a definire.

Il modello di ereditarietà proposto da Java è un modello ad **ereditarietà singola**. A differenza di linguaggi come il C++, in cui una classe derivata può ereditare da molte classi base, Java consente di poter ereditare da una sola classe base come mostrato nella seguente figura.

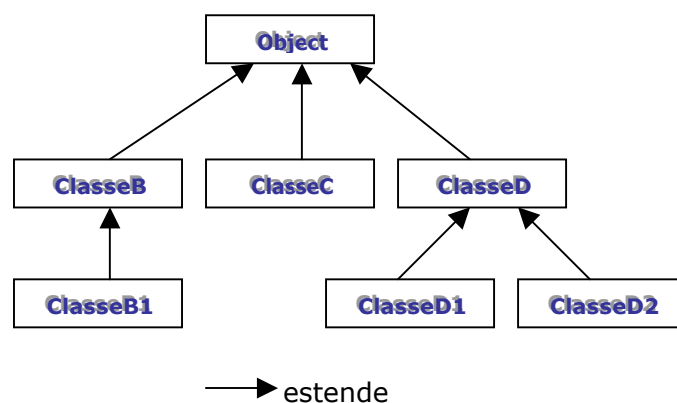


Figura A.2 Modello ad ereditarietà singola in Java

A.4.2 Overload di metodi

Per utilizzare a fondo l'ereditarietà, è necessario introdurre un'altra importante caratteristica di Java: quella di consentire l'**overloading** di metodi. Fare l'overloading di un metodo significa, in generale, dotare una classe di metodi aventi stesso nome ma con parametri differenti.

A.4.3 Estendere una classe base

Definita, per esempio, la classe base Veicolo, sarà possibile definire nuovi tipi di veicoli estendendo la classe generica. La nuova classe, manterrà tutti i dati ed i metodi membro della superclasse con la possibilità di aggiungerne di nuovi o modificare quelli esistenti. La sintassi per estendere una classe a partire dalla classe base è la seguente:

```
class nome_classe extends nome_super_classe.
```

A.4.4 Ereditarietà ed incapsulamento

Nasce spontaneo domandarsi quale sia l'effetto dei modificatori public, private, e protected definiti precedentemente nel caso di classi legate tra di loro da relazioni di ereditarietà. Nella tabella seguente sono schematizzati i livelli di visibilità dei tre modificatori.

Modificatore	Visibilità con sottoclassi
public	Sì
private	No
protected	Sì

Tabella A.2 Modificatori ed ereditarietà

A.4.5 Ereditarietà e costruttori

Il meccanismo utilizzato da Java per assicurare la chiamata di un costruttore per ogni classe di una gerarchia è descritto di seguito.

Primo, ogni classe deve avere un costruttore. Se il programmatore non ne implementa alcuno, Java assegnerà alla classe un costruttore di default con blocco di codice vuoto e senza lista di parametri:

```
public QualcheCostruttore() { }
```

Il costruttore di default viene utilizzato solamente in questo caso. Se viene implementato un costruttore specializzato con lista di parametri di input non vuota, Java elimina completamente il costruttore di default.

Secondo, se una classe è derivata da un'altra l'utente può chiamare il costruttore della classe base immediatamente precedente nella gerarchia utilizzando la sintassi:

```
super(argument_list);
```

dove argument_list è la lista dei parametri del costruttore da chiamare. Una chiamata esplicita al costruttore della classe base deve essere effettuata prima di ogni altra operazione incluso la dichiarazione di variabili. In caso di chiamata implicita a costruttore, Java utilizza una chiamata al costruttore della classe base senza passare argomenti. Se la classe base non è dotata di costruttore senza argomenti verrà generato un errore in fase di compilazione.

A.4.6 Overriding di metodi

Se un metodo ereditato non lavorasse correttamente rispetto a quanto ci si aspetta dalla specializzazione del concetto definito nella classe base, Java consente di riscrivere il metodo originale. Questo significa semplicemente riscrivere il metodo coinvolto all'interno della classe derivata. Anche in questo caso, riscrivendo nuovamente il metodo solo nella nuova classe, non c'è pericolo che la classe base venga manomessa. Il nuovo metodo verrà chiamato al posto del vecchio anche se la chiamata venisse effettuata da un metodo ereditato dalla classe base.

A.4.7 L'oggetto Object

La gerarchia delle classi in Java parte dalla definizione della classe **Object**, che rappresenta la radice dell'albero delle gerarchie. Quando in Java viene creato un nuovo oggetto che non estende

nessuna classe base, Java ne provocherà l'estensione automatica dell'oggetto Object. Questo meccanismo è implementato per garantire alcune funzionalità base comuni a tutte le classi.

Queste funzionalità includono la possibilità di esprimere lo stato di un oggetto in forma di stringhe (tramite il metodo ereditato **toString**), la possibilità di confrontare due oggetti tramite il metodo **equals** e terminare l'oggetto tramite il metodo **finalize**. Quest'ultimo metodo è utilizzato dal garbage collector nel momento in cui elimina l'oggetto rilasciando la memoria, e può essere modificato per gestire situazioni non gestibili dal garbage collector quali referenze circolari.

A.5 Accenno alle Eccezioni

Le eccezioni sono utilizzate da Java in quelle situazioni in cui sia necessario gestire condizioni anomale, ed i normali meccanismi sono insufficienti ad indicare l'errore.

Formalmente, un'**eccezione** è un evento che si scatena durante la normale esecuzione di un programma, causando l'interruzione del normale flusso di esecuzione della applicazione.

Queste condizioni di errore possono svilupparsi in seguito ad una grande varietà di situazioni anomale: il malfunzionamento fisico di un dispositivo di sistema, la mancata inizializzazione di oggetti particolari quali ad esempio connessioni verso database, o semplicemente errori di programmazione come la divisione per zero di un intero. Tutti questi eventi hanno la caratteristica comune di causare l'interruzione dell'esecuzione del metodo corrente.

Il linguaggio Java cerca di risolvere alcuni di questi problemi al momento della compilazione dell'applicazione, tentando di prevenire ambiguità che potrebbero essere possibili cause di errore, ma non è in grado di gestire situazioni di errore complesse o indipendenti da eventuali errori di scrittura del codice.

A.6 Polimorfismo ed ereditarietà avanzata

L'ereditarietà rappresenta uno strumento di programmazione molto potente; d'altra parte il semplice modello di ereditarietà presentato precedentemente non risolve alcuni problemi di ereditarietà molto comuni e, se non bastasse, crea alcuni problemi potenziali che possono essere risolti solo scrivendo codice aggiuntivo.

Uno dei limiti più comuni dell'ereditarietà singola è che non prevede l'utilizzo di una classe base come modello puramente concettuale, ossia priva dell'implementazione delle funzioni. Un altro problema che non viene risolto dal modello di ereditarietà esposto è quello di non consentire ereditarietà multipla, ossia la possibilità di derivare una classe da due o più classi base; la parola chiave **extends** prevede solamente un singolo argomento.

Java risolve tutti questi problemi con due variazioni al modello di ereditarietà definito in precedenza: interfacce e classi astratte. Le **interfacce** sono entità simili a classi, ma non contengono implementazioni delle funzionalità descritte. Le **classi astratte**, anch'esse del tutto simili a classi normali, consentono di non implementare tutte le caratteristiche dell'oggetto rappresentato.

Interfacce e classi astratte permettono di definire un concetto senza dover conoscere i dettagli di una classe posponendone l'implementazione attraverso il meccanismo dell'ereditarietà.

A.6.1 Polimorfismo : "un'interfaccia, molti metodi"

Polimorfismo è la terza parola chiave del paradigma ad oggetti. Derivato dal greco, significa "pluralità di forme" ed è la caratteristica che ci consente di utilizzare un'unica interfaccia per una moltitudine di azioni. Quale sia la particolare azione eseguita dipende solamente dalla situazione in cui ci si trova. Per questo motivo, parlando di programmazione, il polimorfismo viene riassunto nell'espressione "**un'interfaccia, molti metodi**". Ciò significa che possiamo definire un'interfaccia unica da utilizzare in molti casi collegati logicamente tra di loro. Oltre a risolvere i limiti del modello di ereditarietà proposto, Java per mezzo delle interfacce fornisce al programmatore lo strumento per implementare il polimorfismo.

A.6.2 Interfacce

Formalmente, un'interfaccia Java rappresenta un prototipo e consente al programmatore di definire lo scheletro di una classe: nomi dei metodi, tipi ritornati, lista dei parametri. Al suo interno il programmatore può definire dati membro purché di tipo primitivo con un'unica restrizione: Java considererà implicitamente questi dati come static e final (costanti). Quello che una interfaccia non consente è l'implementazione del corpo dei metodi. Una interfaccia stabilisce il protocollo di una classe senza preoccuparsi dei dettagli di implementazione. La sintassi per implementare una interfaccia è la seguente:

```
interface identificatore {
    corpo
}
```

dove **interface** è la parola chiave riservata da Java, identificatore è il nome dell'interfaccia e corpo è una lista di definizioni di metodi e dati membro separati da ";".

Dal momento che una interfaccia rappresenta solo il prototipo di una classe, affinché possa essere utilizzata è necessario che ne esista una implementazione che rappresenti una classe istanziabile. Per implementare una interfaccia, Java mette a disposizione un'altra parola chiave, **implements** con sintassi:

```
class nome_classe implements nome_interfaccia {
    corpo
}
```

Quando una classe implementa una interfaccia è obbligata ad implementarne i prototipi dei metodi definiti nel corpo. In caso contrario il compilatore genererà un messaggio di errore. Di fatto, possiamo pensare ad una interfaccia come ad una specie di contratto che il runtime di Java stipula con una classe. Implementando una interfaccia la classe non solo definirà un concetto a partire da un modello logico (molto utile al momento del disegno dell'applicazione), ma assicurerà l'implementazione di almeno i metodi definiti nell'interfaccia. Conseguenza diretta è la possibilità di utilizzare le interfacce come tipi per definire variabili reference in grado di referenziare oggetti costruiti mediante implementazione di una interfaccia.

Se l'operatore extends limitava la derivazione di una classe a partire da una sola classe base, l'operatore implements ci consente di implementare una classe a partire da quante interfacce desideriamo semplicemente esplicitando l'elenco delle interfacce implementate separate tra loro con una virgola (ereditarietà multipla):

```
class nome_classe implements interface1, interface2,... .interfacen {
    corpo
}
```

Questa caratteristica permette al programmatore di creare gerarchie di classi molto complesse in cui una classe eredita la natura concettuale di molte entità.

Se una classe implementa interfacce multiple, la classe dovrà fornire tutte le funzionalità per i metodi definiti in tutte le interfacce.

A.6.3 Classi astratte

Capitano casi in cui questa astrazione deve essere implementata solo parzialmente all'interno della classe base. In questi casi le interfacce sono restrittive (non si possono implementare solo alcune funzionalità). Per risolvere questo problema, Java fornisce un metodo per creare classi base astratte, ossia classi solo parzialmente implementate.

Le **classi astratte** possono essere utilizzate come classi base tramite il meccanismo della ereditarietà e per creare variabili reference; tuttavia queste classi non sono complete e, come le interfacce, non possono essere istanziate. Per estendere le classi astratte si utilizza, come nel caso di classi normali, l'istruzione extends; di conseguenza solo una classe astratta può essere utilizzata per creare nuove definizioni di classe. Questo meccanismo fornisce un'alternativa alla necessità di dover implementare tutte le funzionalità di una interfaccia all'interno di una nuova classe, aumentando la flessibilità nella creazione delle gerarchie di derivazione.

Per definire una classe astratta Java mette a disposizione la parola chiave **abstract**. Questa clausola informa il compilatore che alcuni metodi della classe potrebbero essere semplicemente prototipi o astratti.

```
abstract class nome_classe
{
data_members
abstract_methods
non_abstract_methods
}
```

Ogni metodo che rappresenta soltanto un prototipo deve essere dichiarato **abstract**. Quando una nuova classe viene derivata a partire dalla classe astratta, il compilatore richiede che tutti i metodi astratti vengano definiti. Se la necessità del momento costringe a non definire questi metodi, la nuova classe dovrà a sua volta essere definita **abstract**.

A.7 Java Thread

Java è un linguaggio multithreaded, cosa che sta a significare che "un programma può essere eseguito logicamente in molti luoghi nello stesso momento". Ovvero, il multithreading consente di creare applicazioni in grado di utilizzare la **concorrenza logica** tra i processi, continuando a condividere tra i thread lo spazio in memoria riservato ai dati.

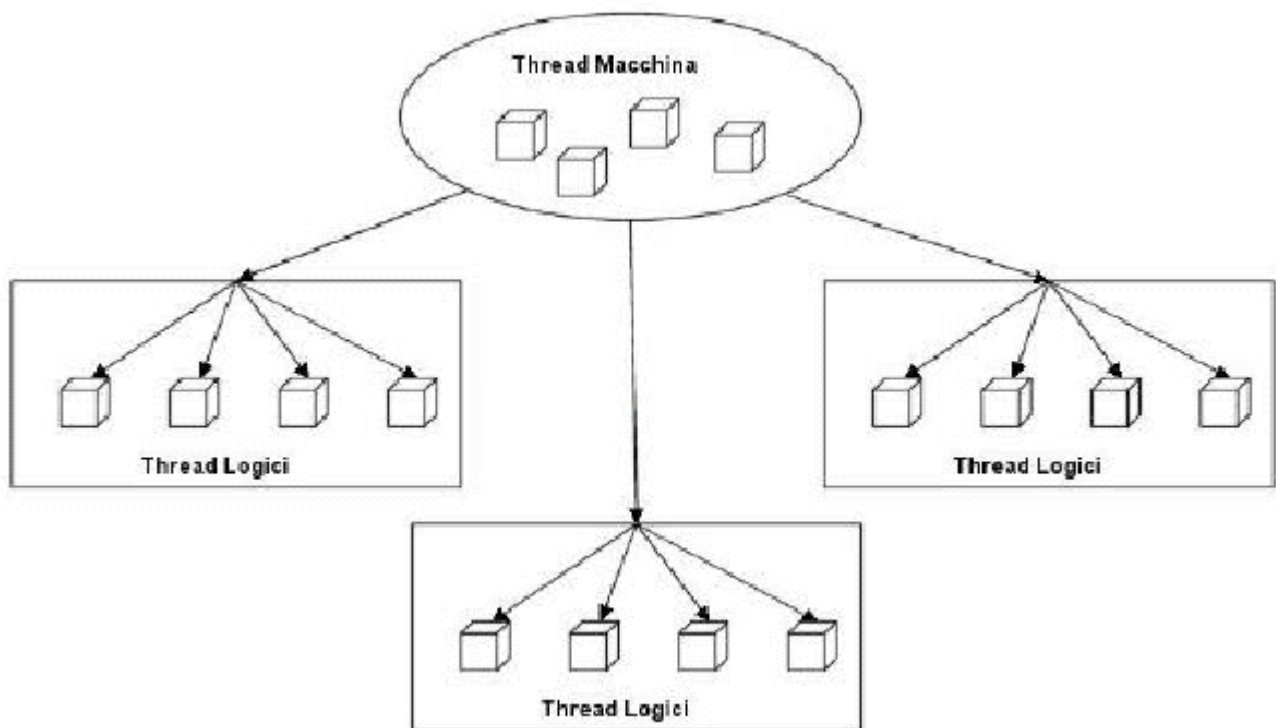


Figura A.3 Concorrenza logica

Nella Figura A.3 viene schematizzato l'ipotetico funzionamento della concorrenza logica. Dal punto di vista dell'utente, i thread logici appaiono come una serie di processi che eseguono parallelamente le loro funzioni. Dal punto di vista dell'applicazione rappresentano una serie di processi logici che, da una parte condividono la stessa memoria dell'applicazione che li ha creati, dall'altra concorrono con il processo principale al meccanismo di assegnazione della CPU del computer su cui l'applicazione sta girando.

A.7.1 Thread di sistema

In Java, esistono un certo numero di thread che vengono avviati dalla JVM in modo del tutto trasparente all'utente. Esiste un thread per la gestione delle interfacce grafiche responsabile della cattura di eventi da passare alle componenti o dell'aggiornamento dei contenuti dell'interfaccia grafica. Il garbage collection è un thread responsabile di trovare gli oggetti non più referenziati e quindi da eliminare dallo spazio di memoria della applicazione. Lo stesso metodo **main()** di una applicazione viene avviato come un thread sotto il controllo della Java Virtual Machine.

Nella Figura A.4 alla pagina seguente sono mostrati i principali thread di sistemi della JVM appena descritti.

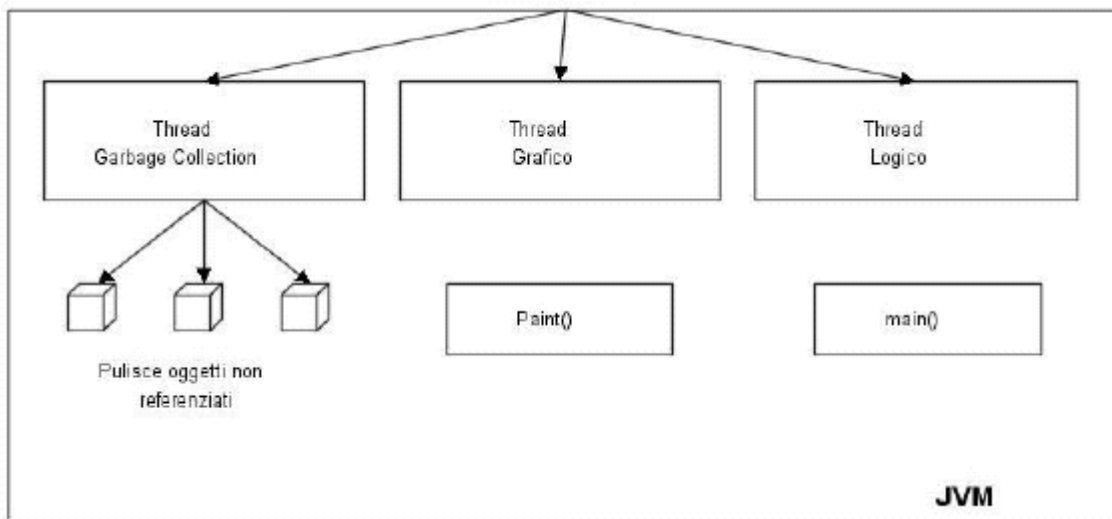


Figura A.4 Schema della struttura dei principali thread di sistema della JVM

A.7.2 La classe java.lang.Thread

In Java, un modo per definire un oggetto thread è quello di utilizzare l'ereditarietà derivando il nuovo oggetto dalla classe base **java.lang.Thread** mediante l'istruzione extends (Figura A.5).

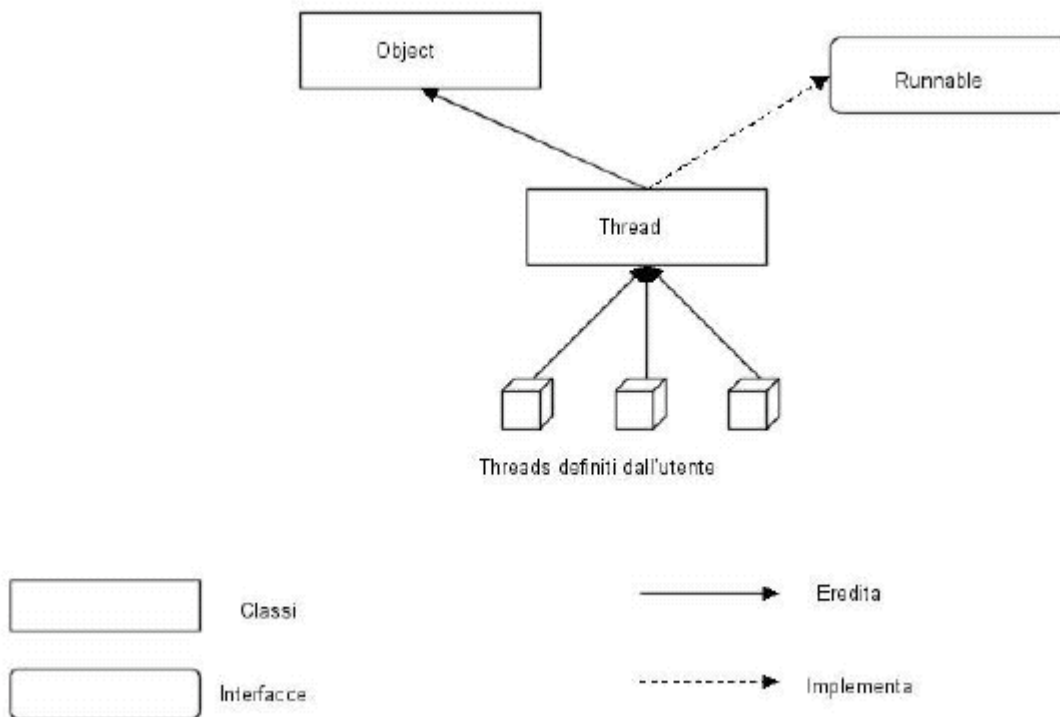


Figura A.5 Definizione di un oggetto thread

La classe `java.lang.Thread` è fornita dei metodi necessari all'esecuzione, gestione e interruzione di un thread. I tre principali sono brevemente descritti di seguito:

- **run** è il metodo utilizzato per implementare le funzionalità eseguite thread. In genere è l'unico metodo su cui effettuare overriding; nel caso in cui il metodo non venga sovrascritto, al momento dell'esecuzione eseguirà una funzione nulla;
- **start** causa l'esecuzione del thread; la Java Virtual Machine chiama il metodo `run` avviando il processo concorrente;
- **destroy** distrugge il thread e rilascia le risorse allocate.

Un esempio di thread definito per ereditarietà dalla classe `Thread` è il seguente:

```
class MioPrimoThread extends Thread {
    int secondi ;
    MioPrimoThread (int secondi) {
        this.secondi = secondi;
    }
    public void run() {
        // ogni slide di tempo definito da secondi
        // stampa a video una frase
    }
}
```

Esempio A.1 MioPrimoThread: un thread di esempio

A.7.3 L'interfaccia Runnable

Ricordando le considerazioni sull'ereditarietà in Java, si è arrivati alla conclusione che l'ereditarietà singola è insufficiente a rappresentare casi in cui è necessario creare gerarchie di derivazione più complesse, e che grazie alle interfacce è possibile implementare l'ereditarietà multipla a livello di prototipi di classe. Supponiamo ora che la classe `MioPrimoThread` sia stata definita a partire da una classe generale diversa da `java.lang.Thread`. A causa dei limiti stabiliti dall'ereditarietà singola, sarà impossibile creare un thread utilizzando lo stesso meccanismo definito nel paragrafo precedente. Sappiamo però che mediante l'istruzione `implements` possiamo implementare più di una interfaccia. L'alternativa al caso precedente è quindi quella di definire un oggetto thread utilizzando l'interfaccia **Runnable** di `java.lang`. L'interfaccia `Runnable` contiene il prototipo di un solo metodo:

```
public interface Runnable
{
    public void run();
}
```

necessario ad indicare l'entry-point del nuovo thread (esattamente come definito nel paragrafo precedente). La nuova versione di `MioPrimoThread` sarà quindi:

```
class MiaClasseBase{
    MiaClasseBase () {
        ...
    }
    public void faiQualcosa() {
        ...
    }
}

class MioPrimoThread extends MiaClasseBase implements Runnable{
    int secondi ;
    MioPrimoThread (int secondi) {
        this. secondi = secondi;
    }
    public void run() {
        // ogni slice di tempo definito da secondi
    }
}
```

```
// stampa a video una frase  
}  
}
```

Esempio A.2 Utilizzo dell'interfaccia Runnable

In questo caso, affinché il thread sia attivato, sarà necessario creare esplicitamente un'istanza della classe Thread utilizzando il costruttore Thread(Runnable r).

```
MioPrimoThread miothread = new MioPrimoThread(5);  
Thread nthread = new Thread(miothread);  
nthread.start() //provoca la chiamata al metodi run di MioPrimoThread;
```


Bibliografia

Documenti principali

- James Gosling, Bill Joy, Guy Steele, Gilad Bracha - **The Java™ Language Specification Second Edition**. Copyright 1996-2000 Sun Microsystems, Inc.
- Bill Shannon - **Java™ 2 Platform Enterprise Edition Specification, v1.3**. Copyright 2001 Sun Microsystems, Inc. (disponibile su <http://java.sun.com/j2ee/docs.html>)
- Danny Coward - **Java™ Servlet Specification Version 2.3**. Copyright 2001 Sun Microsystems, Inc. (disponibile su <http://java.sun.com/products/servlet>)
- Eduardo Pelegrí-Llopart - **JavaServer Pages™ Specification Version 1.2**. Copyright 2001 Sun Microsystems, Inc. (disponibile su <http://java.sun.com/products/jsp>)
- Linda G. DeMichiel, L. Ümit Yalçinalp, Sanjeev Krishnan - **Enterprise JavaBeans™ Specification, Version 2.0**. Copyright 2001 Sun Microsystems, Inc. (disponibile su <http://java.sun.com/products/ejb>)
- Jon Ellis, Linda Ho, Maydene Fisher - **JDBC™ 3.0 Specification**. Copyright 1999-2001 Sun Microsystems, Inc. (disponibile su <http://java.sun.com/products/jdbc>)
- Rahul Sharma - **J2EE™ ConnectorArchitecture Specification Version 1.0**. Copyright 2001 Sun Microsystems, Inc. (disponibile su <http://java.sun.com/j2ee/connector>)
- Nicholas Kassem - **Designing Enterprise Applications with the Java™ 2 Platform, Enterprise Edition**. Copyright 2000 Sun Microsystems, Inc. (disponibile su <http://java.sun.com/j2ee/blueprints>)

Altri documenti

- **Java™ 2 Platform, Enterprise Edition Technical Overview**. Copyright 1998, 1999, Sun Microsystems, Inc. (disponibile su <http://java.sun.com/j2ee/white.html>)
- **Java™ 2 Platform, Standard Edition, v1.3 API Specification**. Copyright 1993-2000, Sun Microsystems, Inc. (disponibile su <http://java.sun.com/j2se/1.3/docs/api/index.html>)
- **Java™ Naming and Directory Interface 1.2 Specification**. Copyright 1998, 1999, Sun Microsystems, Inc. (disponibile su <http://java.sun.com/products/jndi>)
- **Java™ Message Service, Version 1.0.2**. Copyright 1998, Sun Microsystems, Inc. (disponibile su <http://java.sun.com/products/jms>)
- **Java™ Transaction API, Version 1.0.1**. Copyright 1998, 1999, Sun Microsystems, Inc. (disponibile su <http://java.sun.com/products/jta>)
- **Java™ Transaction Service, Version 1.0**. Copyright 1997-1999, Sun Microsystems, Inc. (disponibile su <http://java.sun.com/products/jts>)
- **JavaMail™ API Specification Version 1.1 (JavaMail specification)**. Copyright 1998, Sun Microsystems, Inc. (disponibile su <http://java.sun.com/products/javamail>)
- **JavaBeans™ Activation Framework Specification Version 1.0 (JAF specification)**. Copyright 1998, Sun Microsystems, Inc. (disponibile su <http://java.sun.com/beans/glasgow/jaf.html>)
- **Java API for XML Parsing, Version 1.0 Final Release (JAXP specification)**. Copyright 1999-200, Sun Microsystems, Inc. (disponibile su <http://java.sun.com/xml>)
- **Java™ Authentication and Authorization Service (JAAS) 1.0 (JAAS specification)**. Copyright 1999-2000, Sun Microsystems, Inc. (disponibile su <http://java.sun.com/products/jaas>)
- **Java IDL**. Copyright 1993-99, Sun Microsystems, Inc. (disponibile su <http://java.sun.com/j2se/1.3/docs/guide/idl/index.html>)

- **RMI over IIOP 1.0.1** (disponibile su <http://java.sun.com/j2se/1.3/docs/guide/rmi-iiop/index.html>)
- **The Common Object Request Broker: Architecture and Specification (CORBA 2.3.1 specification)**. Object Management Group (disponibile su <http://cgi.omg.org/cgi-bin/doc?formal/99-10-07>)
- **IDL To Java™ Language Mapping Specification**. Object Management Group (disponibile su <http://cgi.omg.org/cgi-bin/doc?ptc/2000-01-08>)
- **Java™ Language To IDL Mapping Specification**. Object Management Group (disponibile su <http://cgi.omg.org/cgi-bin/doc?ptc/2000-01-06>)
- **Interoperable Naming Service**. Object Management Group (disponibile su <http://cgi.omg.org/cgi-bin/doc?ptc/00-08-07>)
- **The SSL Protocol, Version 3.0** (disponibile su <http://home.netscape.com/eng/ssl3>)
- **A Tutorial on Java Servlets and Java Server Pages** (disponibile su <http://www.apl.jhu.edu/~hall/java/Servlet-Tutorial>)
- **Simplified Guide to the Java™ 2 Platform, Enterprise Edition**. Copyright Information 1999, Sun Microsystems, Inc.
- **Application Server e TP Monitor** (disponibile su <http://www.itware.com>)
- **Application server Java: cosa conta davvero** (disponibile su <http://www.idg.it/networking/nwi2000/Ap129901.htm>)

