Appendice C

Il profilo EJB

Si approfondisce nella presente appendice quanto illustrato per sommi capi al termine del Capitolo 2. Prima di trattare approfonditamente il profilo EJB, è riportata una breve disamina della tecnologia EJB (Enterprise JavaBeans).

Enterprise Java Beans: concetti di base

Benché l'illustrazione dell'architettura EJB esuli completamente dagli obiettivi del presente testo e si tratti di una tecnologia più o meno approfonditamente conosciuta da tutti i tecnici appartenenti alla comunità Object Oriented, è doverosa una brevissima descrizione dei concetti citati più frequentemente. L'architettura EJB è una tecnologia Server-side basata su componenti per lo sviluppo e l'installazione di applicazioni business organizzate in componenti distribuiti. L'utilizzo di tale tecnologia permette di realizzare sistemi scalabili, transazionali, multistrato, sicuri, e così via, senza avere l'incombenza di dover progettare e realizzare una complicata infrastruttura. Ciò rende possibile realizzare rapidamente sistemi distribuiti a oggetti, con rischi notevolmente ridotti e con l'enorme vantaggio che il sistema prodotto, qualora le direttive fornite dalla Sun Microsystem siano rispettate, è in grado di essere eseguito virtualmente su qualsiasi piattaforma che supporti le specifiche EJB (*write once run anywhere*).

Un Enterprise Java Bean è un componente software, operante nel lato server, che può essere distribuito in un ambiente multi-tier. Tipicamente risulta costituito da diversi oggetti, sebbene la struttura interna sia "nascosta" all'oggetto client (altro EJB, Servlet, JSP, Applet, ecc.) per mezzo di opportuna interfaccia. Gli oggetti client non possono richiedere direttamente all'EJB l'esecuzione dei metodi esposti, bensì devono necessariamente passare attraverso un apposito oggetto (detto EJB Object, oggetto EJB). Tale oggetto, pertanto, per poter svolgere il proprio ruolo di mediatore, deve clonare tutti i metodi

business che il componente EJB fornisce, i quali vengono dichiarati esplicitamente dal componente in un'apposita interfaccia denominata Remote Interface (interfaccia remota). L'oggetto cliente, per ovvi motivi, non può neanche istanziare direttamente gli Oggetti EJB (per esempio potrebbero risiedere in un server diverso da quello in cui è in esecuzione il client) e pertanto necessita di acquisirli. Ciò avviene richiedendo la loro "fornitura" a un altro oggetto — detto Home Object che funge da *factory* di EJB Object — il quale è gravato da altre responsabilità oltre alla creazione, come la ricerca e la rimozione degli EJB Object. A questo punto entra in gioco un'altra interfaccia detta Home Interface (interfaccia casa), nella quale devono essere dichiarati i metodi per creare, reperire e distruggere gli oggetti EJB. Il tutto poi è soggetto alla gestione dell'EJB Container (contenitore EJB) il quale fornisce i servizi impliciti ai vari componenti EJB; in altre parole fornisce l'ambiente nel quale i componenti EJB vengono eseguiti. Il Container a sua volta dovrebbe funzionare nell'ambiente fornito da un apposito EJB Server. Tipicamente gli oggetti client dialogano sempre con il Container, il quale interagisce con i vari EJB attraverso opportuni metodi dichiarati nelle interfacce di cui sopra. Si tratta di un vero e proprio mediatore "invisibile", che assolve alle responsabilità di far comunicare i client con gli EJB, coordinare l'esecuzione delle transazioni, provvedere i servizi di persistenza e di sicurezza, gestire il ciclo di vita degli EJB, e svolgere altri compiti ancora.

Esistono tre versioni di EJB: Entity, Session e MessageDriven.

Gli Entity sono utilizzati per memorizzare informazioni che devono persistere alle sessioni utente, e si specializzano in bean a Container Managed Persistence, detti così poiché la gestione della relativa persistenza è demandata al Container, e bean a Bean Managed Persistence, caratterizzati dal possedere al proprio interno la logica per la gestione della persistenza, implementata "manualmente" dallo sviluppatore.

I Session Bean sono utilizzati per modellare la business logic del sistema e quindi realizzano servizi invocabili dai clienti. Esistono due tipi di Session Bean: Stateful e Stateless. I primi sono in grado di memorizzare uno stato tra due invocazioni del client, mentre i secondi non possono farlo e pertanto si prestano a essere utilizzati da più clienti.

I Message Driven Bean sono molto simili ai Session Bean e rappresentano una tipologia di consumatore asincrono di messaggi, invocati dal container al fine di notificare la ricezione di un messaggio JMS (Java Messaging System). Per questo motivo, i Message Driven Bean non necessitano né di Home né di Remote Interface.

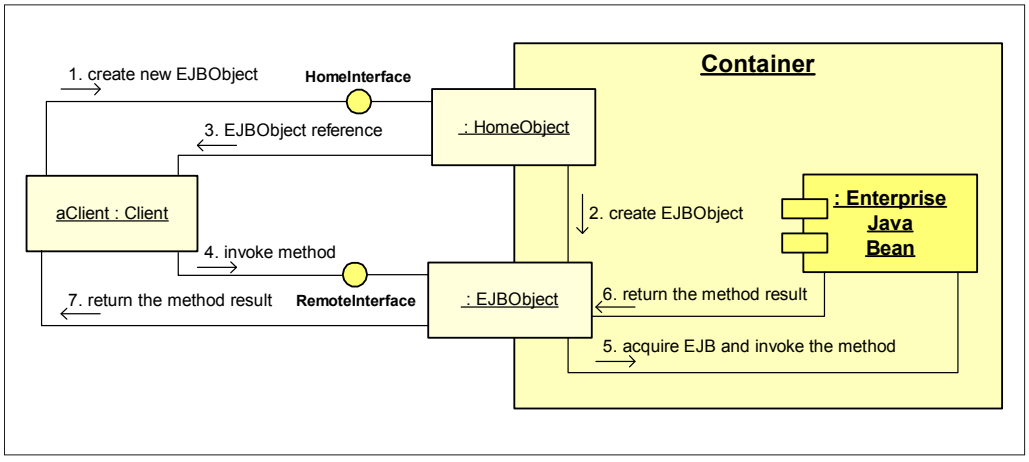
Una volta generate le varie classi che realizzano l'Enterprise Java Bean, le Home e le Remote Interface, il descrittore di dispiegamento (Deployment Descriptor) e le proprietà dell'EJB stesso, è necessario impacchettare tutti questi file in un'unica entità denominata EJB-JAR file, e comprimerlo secondo le direttive del formato ZIP.

Profilo EJB

La formulazione del profilo EJB è il risultato della collaborazione tra Rational, Sun Microsystem e Object Management Group (Java Specification Request JSR-26). Attualmente è disponibile la versione draft (*UML Profile for EJB*, pubblicata nel maggio 2001), il cui principale fautore è Jack Greenfield coadiuvato da un team formato da James Abbott, Loïc Julien, David Frankel, Scott Rich e molti altri ancora. Sebbene di recente formulazione, si tratta di un profilo già obsoleto sia perché basato sulla versione 1.3 delle specifiche ufficiali dello UML, sia perché la versione dell'architettura EJB presa in considerazione è la 1.1. Ciò nonostante si tratta di un documento molto importante e ben congegnato, il cui adeguamento alle ultime versioni delle specifiche UML ed EJB non dovrebbe richiedere eccessivo lavoro. Obiettivo del profilo è definire un mapping standard tra lo UML e l'architettura Enterprise JavaBeans (EJB) (fig. C.1). Il raggiungimento di tale obiettivo comporta:

- la definizione di un approccio standard per la modellazione di sistemi basati sull'architettura EJB e quindi fondati sul linguaggio di programmazione Java;
- il supporto delle esigenze pratiche comunemente incontrate nel disegno di sistemi EJB-based;
- la definizione di una rappresentazione completa, accurata e non ambigua dei manufatti previsti dall'architettura EJB corredati dalla relativa semantica limitatamente ai fini della modellazione;

Figura C.1 — *Versione di un diagramma di collaborazione atto a illustrare il principio di funzionamento dell'architettura EJB. (Rielaborazione, da* ED ROMAN, Mastering EJB, *Wiley).*



- il supporto della costruzione di modelli di "assemblati" EJB frutto della combinazione di package creati utilizzando tool forniti da diversi produttori;

- la semplificazione del disegno di sistemi basati sull'architettura EJB, in modo tale che gli "sviluppatori" da un lato non siano costretti a definire per conto proprio tecniche per la modellazione di sistemi EJB e dall'altro possano confidare in tecniche di reverse engineering compatibili con quanto disegnato;

- il supporto alle diverse fasi del ciclo di vita del software che implicano manufatti attinenti ai componenti EJB (modello di disegno, di sviluppo, deployment e runtime);

- l'attuazione dell'interscambio di modelli di manufatti EJB prodotti per mezzo di tool forniti da diverse ditte produttrici;

- la compatibilità con tutte le API e gli standard del linguaggio di programmazione Java;

- la compatibilità con i profili UML per le analoghe tecnologie, come CORBA e il modello a componenti CORBA

La definizione del profilo EJB, è composto essenzialmente da due parti: il profilo UML vero e proprio e il descrittore UML. Come è lecito attendersi, la prima si occupa della semantica utilizzabile dall'architettura EJB, mentre la seconda introduce un modello UML relativo ai manufatti Java ed EJB che possono essere memorizzati in un file EJB-JAR.

In merito alla prima parte è necessaria una breve considerazione. Dalla lettura del capitolo precedente si è appreso che uno degli obiettivi dello UML è supportare specifiche che risultino indipendenti da particolari linguaggi di programmazione o processi di sviluppo. Ciò chiaramente è molto vantaggioso in quanto rende lo UML idoneo a supportare plausibilmente tutti i linguaggi di programmazione. Però, al tempo stesso, in questo ambito si pone qualche problema giacché l'architettura EJB non è stata basata su un linguaggio di programmazione generale, bensì su uno ben preciso: Java. Ciò finisce per causare un primo ostacolo tra la definizione dello UML e il relativo utilizzo in sistemi EJB-based. In altre parole, la definizione del profilo EJB obbliga a definire, come requisito irrinunciabile, il mapping tra UML e costrutti e package del linguaggio Java utilizzati nell'architettura EJB (fig. C.2). Dall'analisi della fig. C.3, è invece possibile evidenziare come il profilo EJB vero e proprio sia costituito da due componenti, il **modello di disegno** e quello di **implementazione**.

Per quanto riguarda il primo, contiene i diagrammi delle classi che descrivono le interfacce ed eventualmente le classi di implementazione di un Enterprise Java Bean. Il modello di disegno, a sua volta, è suddiviso in due viste: esterna e interna, ciò al fine sia di incoraggia-

Figura C.2 — *Il profilo EJB estende opportune sezioni del profilo Java.*

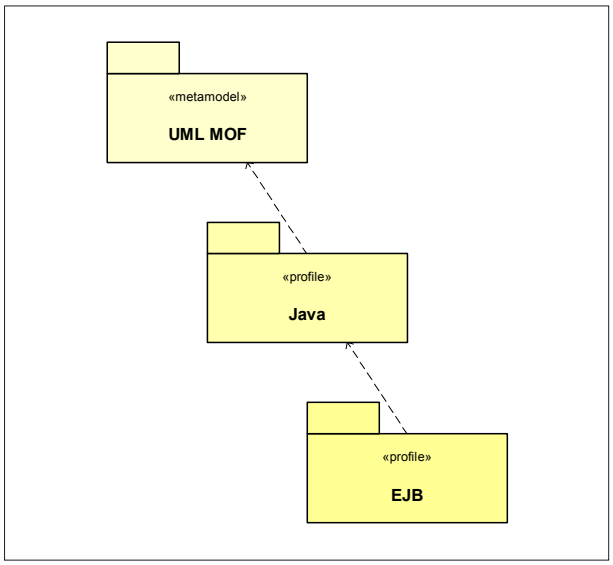
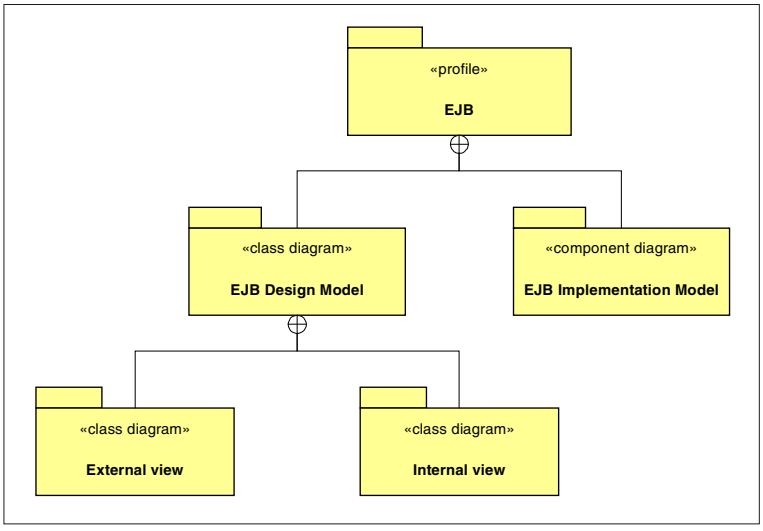


Figura C.3 — *Struttura del profilo EJB. La notazione utilizzata in figura (cerchio con inscritto il simbolo +) rappresenta un'alternativa fornita dallo UML per mostrare relazioni di annidamento tra gli elementi. In questo caso, per esempio, si mostra che il modello EJB Design è incluso nel profilo EJB.*



re a pensare il sistema in termini di componenti già durante le prime fasi del ciclo di vita del software, sia di assecondare la netta separazione tra la specifica di un componente EJB e la relativa implementazione. La componente esterna, come logico attendersi, consente di specificare un EJB dall'esterno ossia dal punto di vista dell'oggetto cliente. A tal fine è importante la definizione delle interfacce `home` e `remote` istanze dell'apposito stereotipo dell'elemento `Class` (`EJB Home Interface` ed `EJB Remote Interface`).

> In questo contesto, quando si parla di stereotipo di un elemento UML, si fa riferimento alla relativa metaclasse appartenente al sottopackage `Core` del package `Foundation` del MOF.

Nell'altra vista, l'attenzione viene spostata alla definizione interna dell'EJB o, se si vuole, alla prospettiva dello sviluppatore. L'EJB viene modellato per mezzo di un opportuno stereotipo dell'elemento UML `Subsystem` contenente sia le interfacce del punto precedente, sia ulteriori stereotipizzazioni dell'elemento `Class` atte a descrivere l'implementazione dell'EJB.

Il modello di implementazione EJB (EJB Implementation Model) contiene diagrammi dei componenti atti a descrivere i manufatti fisici in cui sono memorizzati gli elementi logici di un EJB. In questo ambito viene utilizzato un apposito stereotipo dell'elemento component per rappresentare classi Java, file di risorse e uno stereotipo dell'elemento package per il file `EJB-JAR`.

Riassumendo brevemente, un Enterprise Java Bean viene modellato:

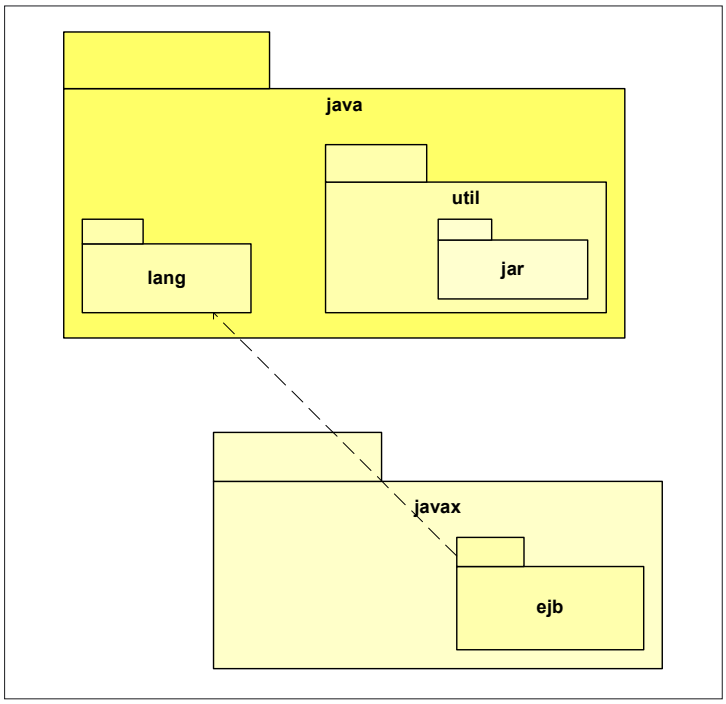
- logicamente o come stereotipo dell'elemento `Class` (nella vista esterna) o dell'elemento `Subsystem` (nella vista interna);
- fisicamente come uno stereotipo dell'elemento `Component`.

Spesso nella definizione formale dei vari profili si incontra il concetto di Virtual MetaModel (VMM, metamodello virtuale), utilizzato per indicare insiemi di estensioni UML. In effetti, ogniqualvolta si introduce un nuovo stereotipo si estende il metamodello base e quindi si finisce per disporre di un nuovo metamodello più potente. In questo contesto quindi ha senso parlare di metamodello virtuale. Ciò, tra l'altro, dovrebbe chiarire anche perché il MOF viene descritto come framework.

Nella pratica, a meno di esigenze di elevato grado di formalità, come la definizione dei profili per esempio, legittimamente i vari stereotipi vengono utilizzati all'uopo senza perdere tempo nel fornirne la definizione rigorosa e tanto meno modificando il metamodello di base dello UML. Ad ogni modo, l'organizzazione del VMM relativo al profilo EJB è rappresentata nella fig. C.4.

[1] Tra i primissimi elementi definiti nel profilo, necessari nella vista esterna del modello di disegno, compaiono le interfacce EJB corredate dai relativi metodi. La definizione di tali

Figura C.4 — *Package del metamodello virtuale definito dal profilo EJB.*



elementi però non sempre può avvenire direttamente, poiché spesso è necessario specificare formalmente gli elementi del linguaggio Java da cui derivano. Pertanto, come già riportato precedentemente, nella definizione formale del profilo EJB è stato necessario specificare tutte, e solo, le parti del linguaggio Java utilizzate dall'architettura EJB.

Per esempio è stato necessario definire formalmente la corrispondenza tra il concetto di classe nel linguaggio Java e quello dello UML. In particolare, la sintassi del costrutto `Class` di Java può essere descritta rigorosamente come segue:

```
[visibility] [modifiers] class Identifier [extends ClassType] [implements TypeList] {
   {ClassMember}
   {InterfaceMember}
   {FieldMember}
   {MethodMember}
}
visibility  = public/proteced/private
modifiers   = [abstract][static][final][strictfp]
TypeList = InterfaceType/ TypeList, InterfaceType
```

Chiaramente una classe Java corrisponde all'equivalente concetto dello UML, però è necessario dar luogo a tutta una serie di mapping, quali:

- la visibilità del linguaggio Java corrisponde all'`ElementOwnership` presente nel Core Package dello UML. In Java quando la visibilità è omessa si intende a livello di package mentre, con la versione 1.4, in UML viene rappresentata attraverso il carattere tilde (~);

- per ciò che concerne i modificatori, si hanno le seguenti corrispondenze:

  - `abstract` corrisponde alla proprietà `isAbstract` dello UML (`GeneralizableElement`);

  - `static` viene introdotto per mezzo del Tagged Value booleano `JavaStatic`; `final` corrisponde alla proprietà `isLeaf` dello UML (`GeneralizableElement`);

  - `strictfp` viene introdotto per mezzo del Tagged Value booleano `JavaStrictfp`;

  - `Identifier` corrisponde ai nomi delle classi UML;

  - `ClassType` si collega ai nomi degli elementi specializzati dall'UML;

  - `InterfaceType` come al punto precedente;

  - `ClassMember` corrisponde alla definizione di classe Java;

  - `InterfaceMember` è relativo alla definizione di interfaccia Java (definita nel profilo);

  - `FieldMember` corrisponde all'elemento Java `Field` (definito nel profilo);

  - `MethodMember` è associato all'elemento Java `Method` (definito nel profilo).

Ovviamente oltre queste corrispondenze, la definizione formale implica la dichiarazione di tutta una serie di vincoli; per esempio una classe Java può specializzare al massimo un'altra classe Java, una classe Java può realizzare un numero qualsiasi di interfacce Java, una classe non può essere dichiarata `final` e `abstract` allo stesso tempo, una classe deve essere definita necessariamente astratta se almeno uno dei suoi metodi è dichiarato astratto, e così via.

Di seguito, l'esame degli stereotipi definiti ritenuti più interessanti. Si tenga presente che l'organizzazione dei package a cui si fa riferimento, a meno di diverse indicazioni, è quella del profilo EJB e non del linguaggio di programmazione Java. Ciò dovrebbe essere evidente anche dall'utilizzo dei due punti come separatore (standard UML).

Nel package `java::lang` è presente lo stereotipo di interfaccia Java (`<<JavaInterface>>`), che specializza quello denominato `<<type>>`: stereotipo predefinito dell'elemento classe.

> Si tratta di uno stereotipo che specifica un dominio di oggetti corredati dalle operazioni ad essi applicabili, senza però definire l'implementazione fisica degli stessi oggetti. In particolare può essere considerato come una particolare versione della classica interfaccia in grado di contenere attributi e associazioni. Obiettivo di queste associazioni è di poter definire il comportamento dei tipi di operazioni senza definire l'implementazione dei dati coinvolti.

Da tener presente che mentre in Java i nomi completamente specificati (*fully-qualified*) dei package si ottengono separando il nome dei package genitori da quello dei figli per mezzo di un carattere "punto" (per esempio `java.lang`), nello UML il separatore è dato da due caratteri "due punti" (`java::ejb`).

La definizione formale dei metodi delle interfacce EJB è ottenuta attraverso i seguenti stereotipi della metaclasse `Operation` appartenenti al package `javax::ejb`: `<<EJBCreateMethod>>`, `<<EJBFinderMethod>>`, `<<EJBRemoteMethod>>`. L'essere stereotipi della metaclasse `Operation` indica che tali elementi possono essere utilizzati per enfatizzare metodi con particolare significato per gli EJB, rispettivamente metodi di creazione, reperimento e remoti.

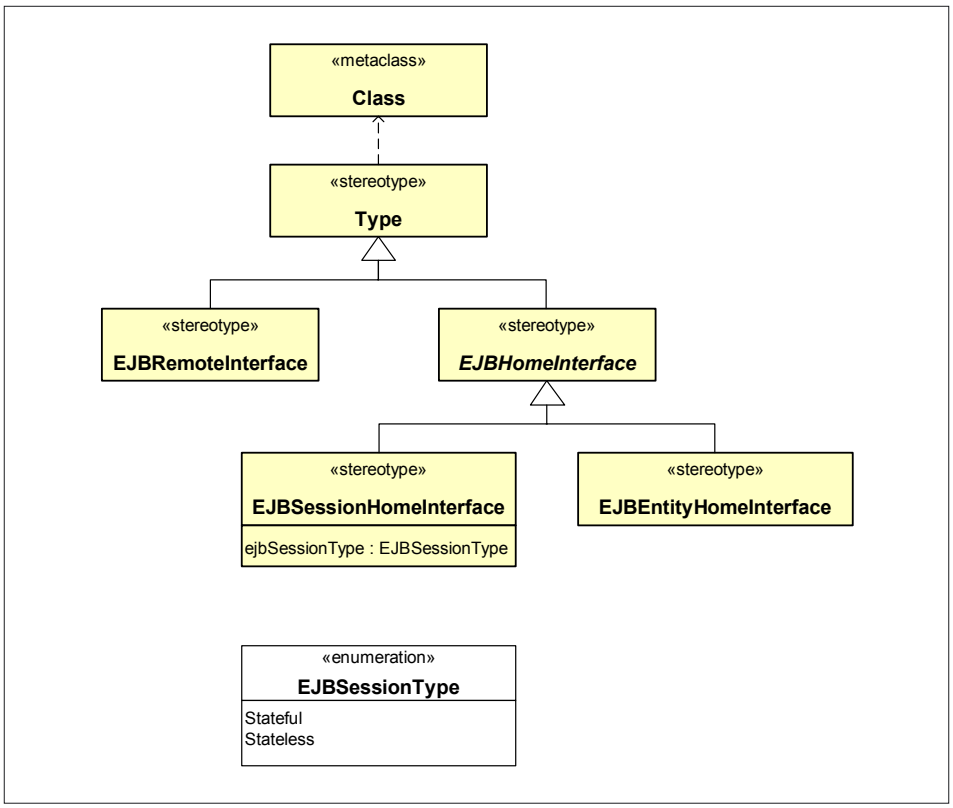
Per quanto concerne le interfacce, sono definite la `<<EJBRemoteInterface>>`, `<<EJBHomeInterface>>`, `<<EJBSessionHomeInterface>>` e `<<EJBEnityHomeInterface>>`.

> Non è presente una definizione di Home Interface per i Message Driven Bean, non solo per questioni di aggiornamento del profilo, ma anche perché tali EJB non dispongono di questa tipologia di interfaccia, così come non dispongono di Remote Interface. Ciò è dovuto al fatto che i Message Driven Bean non possono essere invocati utilizzando un metodo remoto. Essi sono stati introdotti per processare messaggi asincroni e quindi la relativa invocazione deve avvenire da parte di un Messaging Client basato sull'architettura Java Messaging System.

La struttura delle precedenti interfacce è fornita nella fig. C.5.

Per terminare l'elenco degli stereotipi dell'External View, va menzionato `<<EJBPrimaryKey>>`, specializzazione di `Usage`, stereotipo della relazione di dipendenza. Quest'ultimo viene utilizzato per indicare che un elemento necessita di un altro per la completa implementazione o operazione. Nella versione EJB, indica che il fornitore

Figura C.5 — *Interfacce EJB definite nel package* **`javax::ejb`** *(external view). La metaclasse* **`Class`** *appartiene al package* **`Core`***, sottopackage del* **`Foundation`***, del metamodello UML.* **`type`** *è uno stereotipo standard.* **`EJBHomeInterface`** *è una metaclasse astratta.*



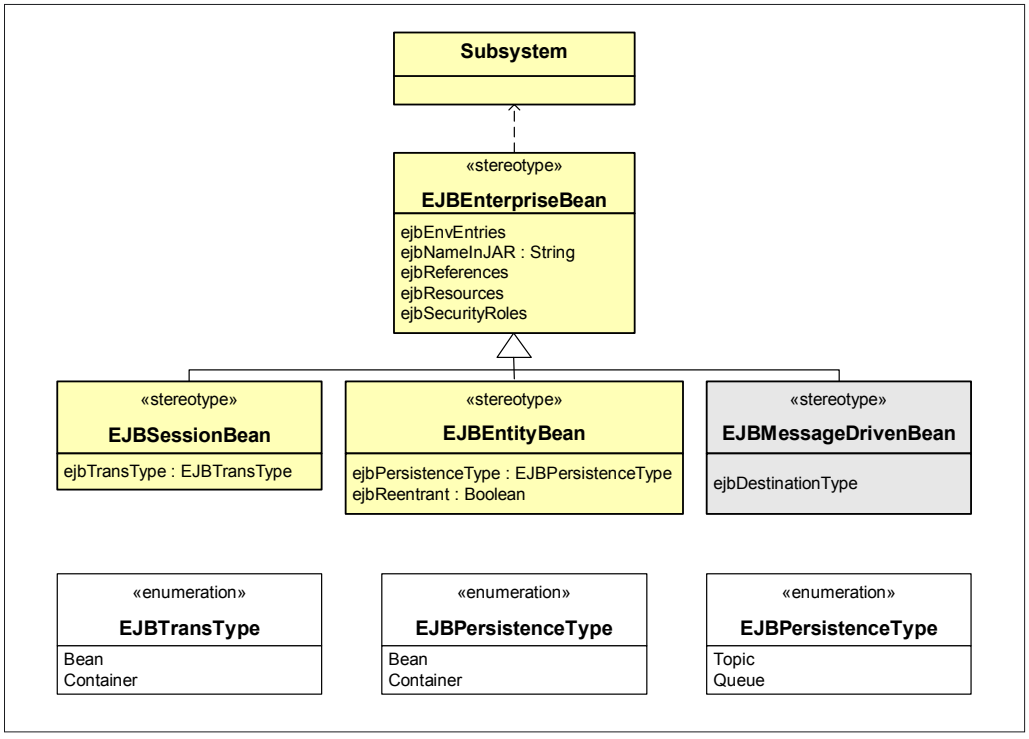
della relazione `Usage` è una classe `EJB Primary Key` necessaria per reperire `EJB Entity Bean`.

Nella vista interna sono presenti diversi stereotipi, il che è abbastanza naturale considerando che il relativo scopo consiste nel descrivere il dettaglio della struttura interna degli EJB. I primi che si incontrano sono gli stereotipi della metaclasse Attributo (`Attribute`), e in particolare la versione "base" `<<EJBCompField>>` il quale evidenzia che il relativo attributo rappresenta un campo gestito dal Container (Container-Managed Field) di un EJB Entity Bean in un contesto di persistenza gestita dal container. Di questo stereotipo è prevista un'ulteriore estensione rappresentata da `<<EJBPrimaryKeyField>>` con significato piuttosto evidente: indica che il campo è una chiave primaria di un EJB Entity Bean nell'ambito di persistenza gestita dal container.

Altri due stereotipi molto interessanti sono quelli etichettati con i nomi `<<EJBRealizeHome>>`, `<<EJBRealizeRemote>>`: si tratta di estensioni della relazione di dipendenza denominata `Abstraction` (Astrazione). Questa relazione è utilizzata per indicare che due o più elementi rappresentano lo stesso concetto a diversi livelli di astrazione. Nel contesto del profilo EJB la relazione è utilizzata per indicare il legame tra interfacce (per esempio `EJB Home Interface`) e relativa implementazione (`EJB Implementation Class`). Quest'ultimo, `<<EJBImplementation>>`, rappresenta a sua volta uno stereotipo, in questo caso dell'elemento `Class`, utilizzato per indicare che la relativa classe implementa un'opportuna interfaccia EJB.

Altri stereotipi molto importanti per l'Internal View sono quelli relativi all'elemento `Subsystem`. In particolare sono presenti gli stereotipi `<<EJBEnterpriseBean>>`, `<<EJBSessionBean>>` e `<<EJBEntityBean>>`, come mostrato in fig. C.6.

Figura C.6 — *Stereotipi dell'elemento* ***`Subsystem`*** *necessari per l'Internal View. Per questioni relative alla versione dell'architettura EJB presa in considerazione (1.1) la specializzazione* ***`EJBMessageDrivenBean`*** *non è ancora disponibile. Gli attributi evidenziati rappresentano Tagged Value descritti successivamente.*



Per terminare l'esame degli stereotipi utilizzati nell'Internal View, si considerino gli elementi `<<EJBReference>>`, specializzazione dello stereotipo `Usage` e `<<EJBAccess>>`, stereotipo della metaclasse associazione (`Association`). Per quanto riguarda il primo non c'è molto da aggiungere, mentre il secondo definisce un ruolo di sicurezza tra un attore UML e un EJB.

Terminata l'analisi degli stereotipi del modello di disegno si passa a esaminare quelli legati al modello di implementazione.

Il primo che si incontra è `<<JavaClassFile>>` (package `java::lang`) che specializza lo stereotipo predefinito `<<file>>`, utilizzato per indicare che un Component è un Java Class File. Un altro stereotipo, questa volta appartenente al package `java::util::jar`, è `<<JavaArchiveFile>>` che serve per indicare che un package rappresenta un JAR.

Ancora, nel package `javax::ejb` è presente lo stereotipo `<<EJB-JAR>>` che specializza `<<JavaArchiveFile>>`, con significato piuttosto intuitivo. Sempre nello stesso package è presente lo stereotipo `<<EJBDescriptor>>`, atto a indicare che un determinato componente rappresenta appunto un EJB Deployment Descriptor. Infine è presente la specializzazione della relazione `Usage`, denominata `<<EJBClientJAR>>` utilizzata per indicare che il "cliente" della relazione rappresenta un `ejb-client-jar` per l'EJB-JAR rappresentato dal fornitore della relazione.

Terminata l'analisi degli stereotipi, è necessario analizzare i vari Tagged Value. Una descrizione esaustiva nel contesto del presente paragrafo risulterebbe piuttosto tediosa e non sempre utile, considerando anche che molti possiedono un significato piuttosto intuitivo. Pertanto, di seguito, l'attenzione viene focalizzata esclusivamente sui Tagged Value ritenuti più interessanti e/o ricorrenti.

Per esempio nel package `java::lang` sono definiti i valori etichettati `JavaNative` e `JavaThrows`, entrambi applicabili agli stereotipi della metaclasse `Operation`. Il primo è di tipo booleano e serve ad indicare se un metodo è nativo Java o meno, mentre il secondo è un array di stringhe atto a riportare la lista dei nomi delle eccezioni, separate dal carattere virgola (*comma-delimited*), scatenabili da un metodo. Un altro esempio di Tagged Value, definito sempre nello stesso package, è `JavaFinal` il quale con il suo valore booleano è utilizzato per indicare se un parametro è modificabile (`false`) o meno.

Da tener presente che, ogniqualvolta compare un Tagged Value di tipo booleano, la convenzione sancisce che la relativa presenza implica un valore `true` (riportare `JavaFinal = true` o semplicemente `JavaFinal` è completamente equivalente), mentre l'omissione indica un valore `false`. Nell'analisi della vista esterna del modello di disegno EJB sono stati introdotti alcuni Tagged Value, come per esempio:

- `EJBSessionType` di tipo enumerato, associato allo stereotipo `<<EJBSessionHomeInterface>>`, il quale indica la tipologia del relativo Session Bean: `stateful` o `stateless`;

- `EJBNameInJar`, il quale indica il nome utilizzato dell'EJB quando viene impacchettato nel file EJB-JAR. Tipicamente si utilizza il nome dell'EJB Remote Interface;

- `EJBEnvEntries`, associato a qualsiasi EJB, ne indica, attraverso una lista di tuple (nome, tipo, valore) separate da virgola, le *environment entries*;

- `EJBResources`, in maniera del tutto simile al Tagged Value precedente, indica le Resource Factory dell'EJB;

- `EJBReferences`, associato a qualsiasi EJB, ne specifica, attraverso una lista di tuple (nome, tipo, home, remote) separate da virgola, gli altri EJB referenziati;

- `EJBSecurityRoles`, associato a qualsiasi EJB, ne indica, attraverso una lista di tuple (nome, link) separate da virgola, il nome del ruolo di sicurezza in grado di invocare i metodi;

- `EJBTransType`, dichiara se la gestione delle transazioni di un Session Bean è demandata al Container oppure è eseguita direttamente dal Bean;

- `EJBPersistenceType`, si tratta di un tipo enumerato assolutamente analogo al precedente con l'unica differenza che si applica agli Entity Bean e non ai Session;

Un altro Tagged Value molto interessante è denominato `EJBTransAttribute`, applicabile ai metodi. Esso sancisce la policy utilizzata per la gestione delle transazioni. In particolare i valori previsti dal tipo enumerato sono: `NotSupported`, `Supports`, `Required`, `RequiredNew`, `Manadatory` e `Never`. Brevemente il significato degli attributi transazionali è il seguente:

- Non supportata (`NotSupported`), indica che l'EJB non può essere coinvolto in transazioni. Pertanto quella eventualmente attiva viene sospesa;

- Mai (`Never`), specifica che l'EJB non può essere assolutamente coinvolto in transazioni. Pertanto se all'atto dell'invocazione esiste una transazione attiva, l'EJB stesso genera un'eccezione;

- Richiesta (`Required`), l'EJB deve essere sempre eseguito nel contesto di una transazione. Se all'atto dell'invocazione è presente una transazione attiva, allora l'EJB la utilizza, altrimenti ne inizia una nuova;

Figura C.7 — *Vista esterna del modello di disegno di un Enterprise Session Bean relativo a una sessione utente.*

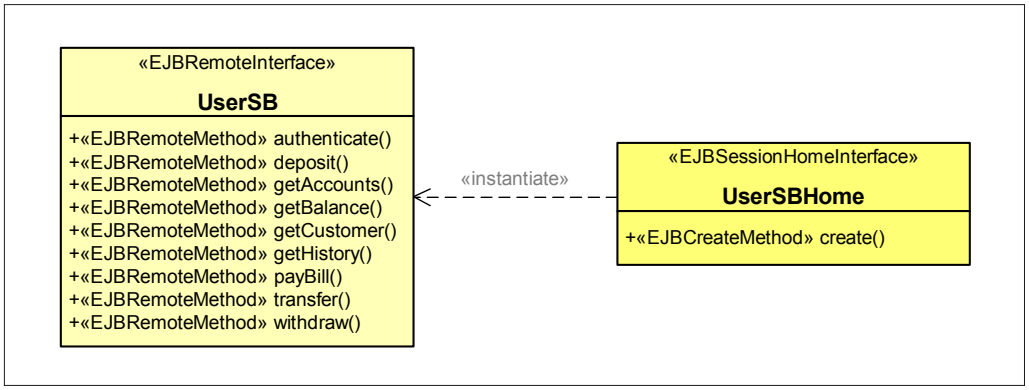
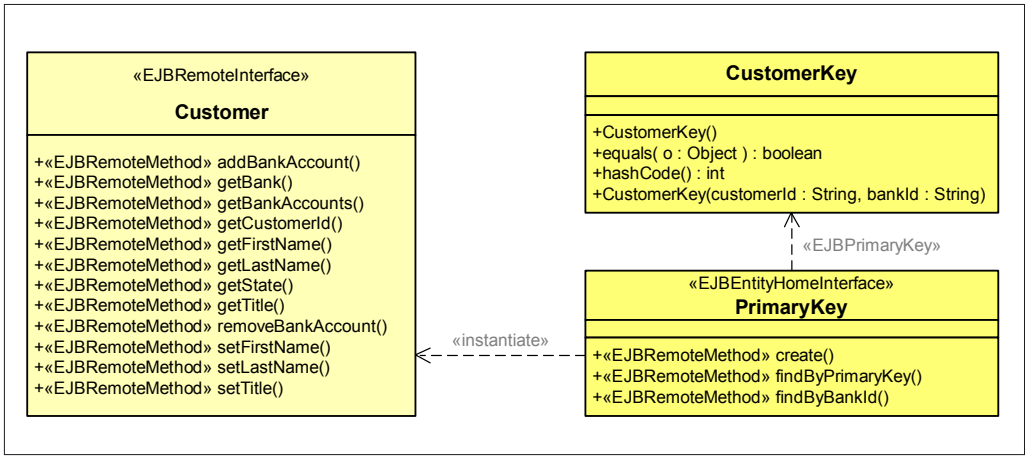


Figura C.8 — *Vista esterna del modello di disegno di un Enterprise Entity Bean dedicato ai dati dei clienti.*



- Richiesta nuova (`RequiredNew`), l'EJB deve essere sempre eseguito nel contesto di una nuova transazione;

- Supportata (`Supports`), sancisce che l'EJB è del tutto indifferente alle transazioni. Se ne esiste una attiva, il relativo stato rimane inalterato, altrimenti può proseguire nell'esecuzione senza transazioni attive;

- Obbligatoria (`Mandatory`), richiede che una transazione sia già attiva prima dell'invocazione dell'EJB. In caso contrario un'eccezione viene scatenata.

A questo punto è finalmente giunta l'ora di riassumere i concetti esposti in chiave operativa attraverso il ricorso ai tanto amati esempi.

Il sistema preso in considerazione è relativo ad un'ipotetica banca. Più precisamente si tratta del modello IBM ITSO Bank, riportato nel profilo EJB, utilizzato spesso anche dalla stessa IBM per illustrare ambienti runtime IBM e vari tool.

Il primo modello considerato è relativo alla Design View. Si inizi con il focalizzare l'attenzione sulla vista esterna e in particolare su un primo componente dedicato alla sessione utente. Come è lecito attendere si tratta di un Enterprise Session Bean, del quale viene fornita la vista esterna, quella a cui sono interessati gli oggetti "cliente" (fig. C.7). Come si può ben notare, l'attenzione viene focalizzata esclusivamente sulla definizione delle interfacce, a tal fine sono utilizzati gli stereotipi `<<EJBRemoteInterface>>` e `<<EJBSessionHomeInterface>>`. Per ciò che concerne i metodi, gli stereotipi utilizzati sono `<<EJBRemoteMethod>>` e `<<EJBCreateMethod>>`.

Nel secondo esempio viene presentata la External View di un Enterprise Entity Bean atto a memorizzare i dati dei clienti della banca (fig. C.8). A differenza del caso precedente, lo stereotipo utilizzato per la Home Interface è la specializzazione `<<EJBEntityHomeInterface>>` proprio per indicare che si tratta di un Enterprise Entity Java Bean. Sono poi presenti gli stereotipi `<<EJBFinderMethod>>` e `<<EJBCreateMethod>>` al fine di enfatizzare la natura dei relativi metodi. Infine è presente lo stereotipo `<<EJBPrimaryKey>>` della relazione di dipendenza, per mostrare che l'oggetto `CustomerKey` fornisce appunto la chiave primaria per il reperimento degli Entity Bean relativi ai clienti (*customer*).

A questo punto si passa all'Internal View. Da questo punto in poi viene preso in considerazione esclusivamente l' Entity Bean mostrato nella fig. C.8. Come già illustrato precedentemente, sebbene possa sembrare piuttosto singolare, la vista interna degli EJB viene modellata utilizzando l'elemento `Subsystem` anziché `Component`.

> La metaclasse `Subsystem`, insieme a `Model` e `Package` (genitore degli altri due), rappresenta l'elemento di maggiore interesse del package `Model Management`, dipendente dal `Foundation`, del metamodello UML. In prima analisi, tali elementi sono utilizzati per raggruppare unità, in qualche modo relazionate, necessarie ad altri `ModelElements`. In particolare ci si avvale dell'elemento `Model` per rappresentare diverse viste di uno stesso sistema. L'elemento `Package` è utilizzato, in modo del tutto intuitivo, nel contesto di un modello per raggruppare elementi di modellazione. Infine `Subsystem` permette di rappresentare ben definite unità comportamentali di un sistema.
>
> Come accennato poc'anzi, la metaclasse `Subsystem` specializza quella `Package`. Mentre quest'ultima rappresenta un meccanismo generico per organizzare gli elementi del modello, un

Figura C.9 — *Vista interna del modello di disegno relativa all'Entity Bean Customer.*

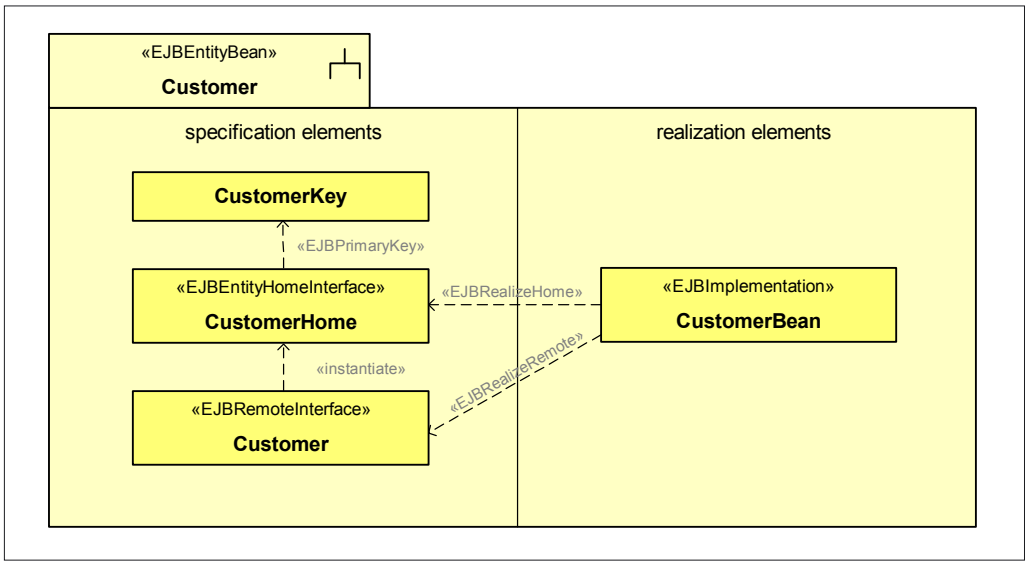


Figura C.10 — *Illustrazione dell'utilizzo dello stereotipo* ***<<EJBReference>>.***

Subsystem tratteggia un'unità comportamentale di un sistema nel modello. L'elemento `Subsystem` però non eredita unicamente da `Package`, ma anche da `Classifier`. Ciò gli permette di offrire interfacce, operazioni, ecc. Il suo contenuto è partizionato in due sezioni: elementi di specificazione e di realizzazione. Nella prima trovano posto le operazioni nel sottosistema, così come Use Case, State Machines, ecc. ossia tutti elementi che permettono di fornire elementi di specifica.

Tipicamente un sottosistema viene rappresentato utilizzando la stessa notazione dell'elemento `Package` con in più il simbolo di fork. Chiaramente, nulla vieta di rappresentarlo come un `Package` con lo stereotipo `<<subsystem>>`.

Si tratta di un elemento dello UML di utilizzo non frequente, e pertanto non molto conosciuto dai disegnatori. Verosimilmente questa scelta è dovuta alla definizione limitata di componente presente nella versione 1.3 dello UML. Già con la versione 1.4 le cose sono cambiate parecchio, quindi, alla luce delle nuove release dello UML, probabilmente il profilo EJB subirà importanti aggiornamenti.

Come si può notare dal diagramma di fig. C.9, l'elemento `Subsystem` si presta a essere utilizzato per descrivere i componenti EJB, in quanto dotato di una sezione relativa alle specifiche (Specification Elements) e di una efferente dalla realizzazione (Realization Elements). Il problema è che, al momento in cui viene scritto questo testo, non tutti i tool commerciali permettono di rappresentare sottosistemi secondo le direttive UML.

Figura C.11 — *Vista interna dell'Entity Bean* ***`Customer`***.

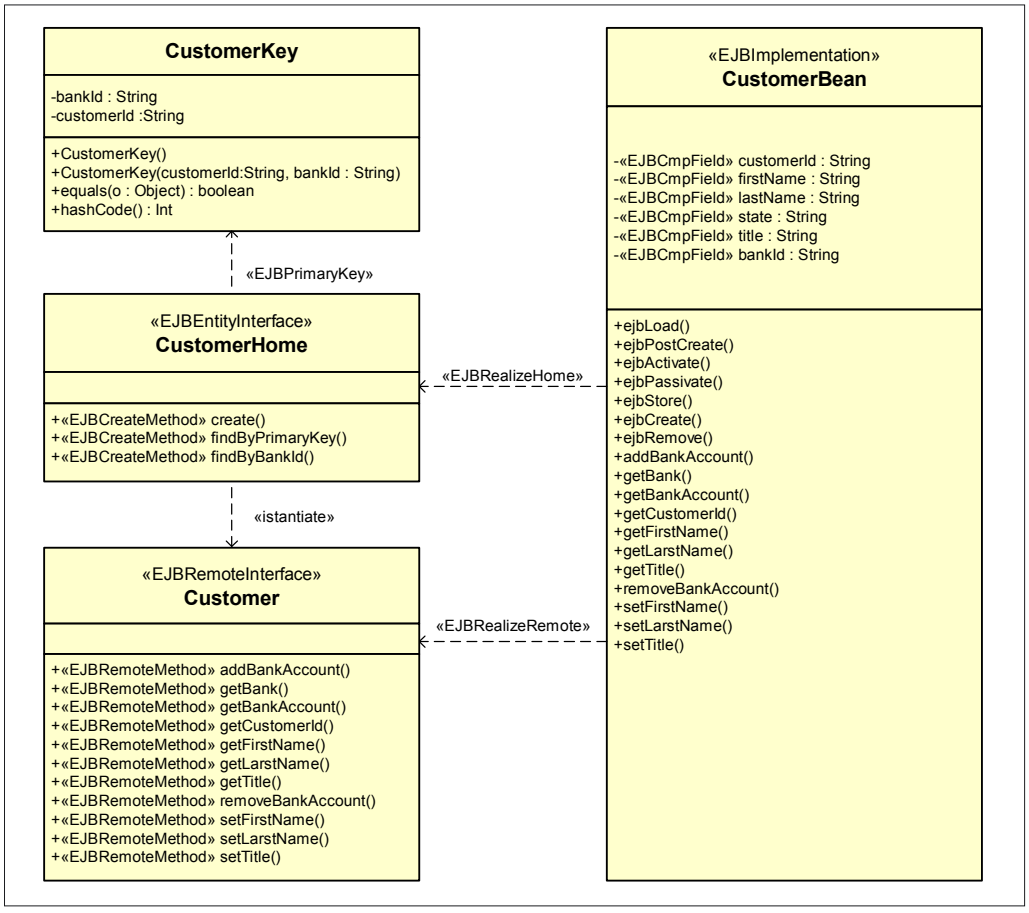
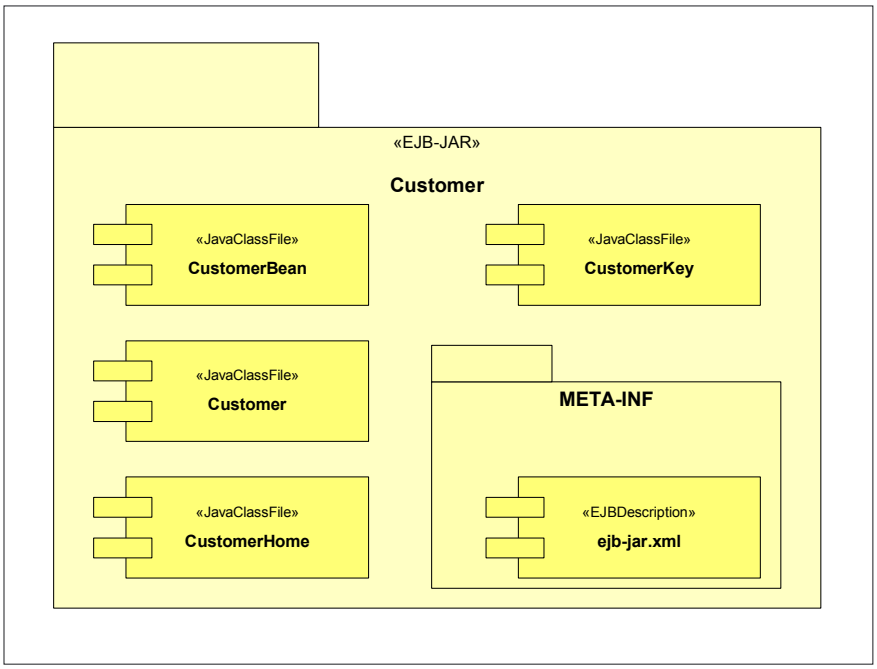


Figura C.12 — *Implementation View dell'Entity Bean* **`Customer`**.



Lo stereotipo `<<EJBReference>>`, specializzazione di `Usage`, stereotipo della relazione di dipendenza, evidenzia la necessità di un EJB di accedere a un altro EJB durante la propria esecuzione. Per esempio la fig. C.10 mostra che l'Entity EJB `Customer` necessita di accedere all'EJB `BankAccount`.

A questo punto è giunto il momento di rappresentare l'implementazione interna dell'Entity Bean. L'elemento introdotto nel diagramma di fig. C.11 è lo stereotipo `<<EJBCmpField>>` nella sezione degli attributi della classe `CustomerBean` la cui descrizione è stata riportata precedentemente.

Per concludere l'illustrazione dell'utilizzo dei vari stereotipi presentati nella prima parte, è necessario passare al modello di implementazione. Il primo stereotipo utilizzato è `<<EJB-JAR>>`, specializzazione `<<JavaArchiveFile>>`, stereotipo dell'elemento `Package`, all'interno del quale vengono raggruppati tutti file contenenti le classi Java con in più l'aggiunta del descrittore dell'EJB (Deployment Descriptor). Il tutto viene esemplificato in fig. C.12.

Da questa breve trattazione emerge che, sebbene il profilo EJB sia ancora in uno stadio piuttosto evolutivo — è necessario un processo di adeguamento sia per incorporare le direttive della versione 1.4 dello UML, sia per allinearlo alla versione 2.0 dell'architettura

EJB — si tratta nel suo complesso di un ottimo strumento di supporto alla progettazione di sistemi Component-based utilizzanti l'architettura EJB.

Il profilo non solo risolve il problema della rappresentazione dei manufatti (Artifact) previsti dall'architettura EJB, ma fornisce anche direttive su come affrontare il processo di sviluppo di sistemi Component-based. A tal fine il profilo si adegua ai processi di sviluppo dei sistemi, organizzando i vari manufatti previsti dalle diverse fasi, in opportuni modelli; in particolare sono previsti i modelli di disegno e implementazione.

Il profilo è focalizzato essenzialmente sulle fasi di disegno e non su quelle di specifica. Il motivo è piuttosto ovvio: queste ultime devono essere generiche e non fondate su specifiche architetture come l'EJB: chiaramente devono assumere che il sistema verrà sviluppato secondo un particolare paradigma, ma non entrare assolutamente nei dettagli tecnici. Lo stesso discorso non può essere applicato al modello di disegno che, per forza di cose, deve essere realizzato in base alle direttive fornite dalla particolare architettura selezionata.

Molto apprezzata è la suddivisione del modello di disegno nelle viste esterna e interna. Ciò risulta molto utile nel contesto di processi di sviluppo del software iterativi e incrementali: nelle prime iterazioni della fase di elaborazione è possibile concentrarsi sulla definizione del comportamento dei vari componenti e sulle relazioni con gli altri, rimandando a successive iterazioni i dettagli interni.

La definizione formale di tale profilo permette di mantenere standard l'utilizzo dello UML anche per tecnologie specifiche quali l'architettura EJB. Ciò è molto importante non solo in chiave comunicativa (possibilità di far circolare più agevolmente i vari modelli, semplificazione del processo di inserimento di nuove risorse, ecc.) ma anche come direttiva per le aziende fornitrici dei vari tool (garanzia che il reverse engineering sia consistente con il normale processo, forward engineering, possibilità di comunicazione tra i modelli prodotti con tool diversi ecc.).

Qualche perplessità permane per ciò che concerne il dettaglio di vari manufatti. Sebbene le funzioni di reverse engineering potranno fornire un notevole contributo nello snellire il lavoro; la rappresentazione di tutti i vari elementi potrebbe comunque risultare gravosa.

Un'altra constatazione è che la stessa definizione del profilo è, se c'è ne fosse ancora bisogno, l'ennesima dimostrazione dell'efficacia, dell'eleganza e della flessibilità del metamodello UML. In effetti è stato possibile specializzarlo, dando luogo al famoso VMM, senza dover assolutamente modificare l'architettura di base.

Il profilo EJB rappresenta indubbiamente un primo passo nella direzione della definizione di standard di notevole importanza nella progettazione di sistemi EJB-based.