Reingegnerizzazione ed implementazione del modulo di aggregazione dati di contesto dell'architettura CARE

Marco Fornoni

A mio padre e mia madre, Dino e Teresa e a mia sorella Valeria.

Indice

Introduzione

Nel campo delle scienze dell'informazione un'esigenza molto sentita ai giorni nostri è l'accessibilità dei servizi distribuiti tramite Internet in uno scenario in cui i servizi stessi vengono fruiti da dispositivi sempre più eterogenei. Questo allargamento dello spettro di dispositivi in grado di collegarsi alla rete e la convergenza delle tecnologie di comunicazione ed intrattenimento riguarda anche e soprattuto i dispositivi così detti "mobili", le cui caratteristiche principali sono la limitatezza e la variabilità delle risorse a disposizione. I dispositivi mobili, quali ad esempio telefoni cellulari e palmari, richiedono che i contenuti dei servizi fruiti siano adattati dinamicamente in base al contesto e alla locazione degli utenti e alle caratteristiche dell'ambiente in cui operano.

Per supportare questa classe di servizi è necessario utilizzare una architettura per la gestione dei dati di profilo degli utenti e dei loro dispositivi. In quest'ambito il laboratorio DaKWE (Data Knowledge and Web Engineering) ha progettato ed implementato una architettura distribuita per la gestione di dati di contesto in grado di supportare la dinamica dei dati tramite politiche di adattamento. Lo scopo della presente tesi è la reingegnerizzazione e l'implementazione in linguaggio Java del modulo per l'aggregazione di dati di contesto distribuiti, basato su criteri di efficienza e aderenza al formalismo standard CC/PP. Basandosi su di un'analisi dell'architettura e sulla scelta tra possibili soluzioni alternative, la parte principale del lavoro di tesi è stata quindi la reingegnerizzazione e reimplementazione dei moduli di aggregazione dei profili e delle politiche di adattamento, e delle comunicazioni tra i moduli dell'architettura. In particolare, tutto il lavoro implementativo, discostandosi parecchio dalla soluzione precedente, è stato svolto senza considerare l'implementazione tecnica precedente ma considerando ed affrontando il problema come ex-novo e cercando di volta in volta la soluzione più efficiente possibile.

La trattazione è suddivisa in due capitoli:

1. Nel primo capitolo viene presentata l'architettura esistente, fornendo una descrizione di come questa sia stata progettata ed implementata. Vengono poi analizzati i moduli della suddetta architettura che hanno richiesto l'intervento oggetto di questa tesi, motivando la necessità di una loro reingegnerizzazione.

2. Il secondo capitolo contiene la descrizione del processo di reingegnerizzazione del modulo di aggregazione dati. La trattazione segue l'ordine temporale in cui sono state effettuate le diverse operazioni. Nella sezione 2.1 è presentata la progettazione ed implementazione di una libreria per la gestione di profili CC/PP e delle politiche di adattamento. Nella sezione 2.2 la discussione verte invece sulla reingegnerizzazione dei moduli *Profile Manager* aventi il compito di gestire i profili CC/PP e le politiche di adattamento associate agli utenti. Nella sezione 2.3 è quindi presentato il lavoro di reingegnerizzazione del modulo di aggregazione dei profili CC/PP e delle politiche di adattamento, mentre l'ultima sezione 2.4 contiene invece uno studio di diverse possibili alternative per la comunicazione tra il modulo di aggregazione dati ed i *Profile Manager*.

Capitolo 1

Architettura esistente e aree di intervento

In questo capitolo sarà brevemente esposta ed analizzata l'implementazione dell'architettura precedente al lavoro di tesi. Nella descrizione si cercherà di mantenere un livello di astrazione adeguato alla comprensione degli argomenti, tralasciando gli aspetti tecnici che esulano da questa trattazione e specificando meglio quelli più rilevanti. Infine saranno evidenziati alcuni punti critici dell'architettura che sono alla base del lavoro di tesi.

1.1 L'architettura esistente

Il linguaggio che è stato scelto per l'implementazione dell'architettura è Java, principalmente per le sue caratteristiche di portabilità. Solamente ove imposto da requisiti di efficienza o di altro tipo si è passati ad altre soluzioni. La comunicazione tra moduli interni all'architettura è stata affidata al paradigma dei Web Services e la persistenza dei dati al database MySQL.

Il framework progettato è composto principalmente da tre entità: l'utente ed il suo dispositivo, l'operatore di rete con la sua infrastruttura ed il service provider con la sua infrastruttura. Ad ognuna di queste entità è associato un

Profile Manager, chiamato rispettivamente: *User Profile Manager*, *Operator Profile Manager*, *Service Provider Profile Manager*.

I moduli *Profile Manager* Come descritto in [1], ogni profile manager ha lo scopo di gestire, per l'entità specificata, i dati di contesto dell'utente conosciuti dall'entità stessa, garantendone la persistenza e l'aggiornamento. È importante notare che l'architettura è stata pensata per gestire un numero arbitrario di entità. Ogni profile manager implementa la persistenza dei dati degli utenti mediante una base di dati MySQL. La comunicazione tra i *Profile Manager* ed il *Context Provider* è realizzata mediante la tecnologia dei Web Services e ogni *Profile Manager* espone il proprio Web service cui il *Context Provider* inoltra le richieste. Ogni volta che viene effettuata una richiesta al Web service, il *Profile Manager* esegue una connessione al database, effettua la query per recuperare il profilo, effettua una serializzazione XML di tale profilo ed invia infine come risposta la stringa così serializzata al *Context Provider*. La gestione delle politiche è affidata ad un RuleML Repository (vedi [2] e [3]).

Il modulo *Context Provider* La parte centrale dell'architettura è rappresentata dal modulo *Context Provider*, il cui scopo principale è quello di creare e mantenere aggiornato un profilo unico associato all'utente. Per fare ciò, si occupa di:

- recuperare i profili parziali dell'utente dai *Profile Manager*;
- effettuare l'aggregazione degli stessi sulla base di direttive di priorità;
- effettuare delle inferenze sulla base delle politiche stabilite dai vari *Profile Manager*;

In un'architettura distribuita in cui ogni entità è in grado di fornire un profilo relativo ad un utente, la possibilità che due entità forniscano dei profili in conflitto tra loro è elevata (vedi [5]). Esiste quindi la necessità di progettare un sistema per risolvere univocamente tutti i conflitti che possono

verificarsi. Nell'architettura progettata, la risoluzione dei conflitti e quindi l'aggregazione dei profili è affidata al modulo *Merge* che, a sua volta, esegue le azioni stabilite delle direttive di aggregazione che permettono la risoluzione univoca dei numerosi conflitti che possono verificarsi durante l'aggregazione dei profili stessi.

Un tipico esempio di conflitto si ha quando sia l'*Operator Profile Manager*, che monitora la banda disponibile, sia l'*User Profile Manager* che esegue la stessa operazione lato Client, forniscono due valori diversi per l'attributo *AviableBandwidth*, generando così un'ambiguità che è necessario risolvere. Per risolvere questo tipo di conflitti vengono definite dal fornitore del servizio le direttive che assegnano delle priorità agli attributi provenienti dai diversi *Profile Manager*. Il modulo *Merge* si occupa dell'applicazione sistematica delle direttive di risoluzione dei conflitti.

L'architettura attuale permette inoltre di impostare delle politiche (policy) su ogni *Profile Manager* (in particolare su *SPPM* e *UPM*) che permettono di derivare il valore degli attributi sulla base del valore di quelli provenienti dal modulo *Merge*. Di questa operazione si occupa il modulo *Inference Engine*. Le direttive per la risoluzione dei conflitti permettono di impostare la priorità dei Componenti/Attributi, mediante una appropriata sintassi. Ad esempio:

1. setPriority */* = (SPPM,UPM,OPM)
2. setPriority NetSpecs/* = (OPM,UPM,SPPM)
3. setPriority UserLocation/Coordinates (UPM, OPM)

Con la direttiva *1* si intende stabilire che, per tutti i Componenti / Attributi l'*SPPM* ha priorità maggiore rispetto ad *UPM* ed *OPM* La direttiva *2* specifica che per tutti gli attributi appartenenti al Componente NetSpecs la priorità è nell'ordine: 1)*OPM* 2)*UPM* 3)*SPPM*. La direttiva *3* stabilisce che per l'attributo Coordinates contenuto nel componente UserLocation l'*UPM* ha priorità maggiore sull'*OPM*, mentre l'assenza dell'*SPPM* sta ad indicare

che, per questo attributo, valori provenienti dall'*SPPM* non devono mai essere considerati. Le direttive hanno priorità decrescente in base all'ordine in cui sono dichiarate. Infatti lo scopo della prima direttiva quello di stabilire una regola generale per l'aggregazione dei profili, per tutti quegli attributi e componenti per cui più sotto non sia definita una regola ad-hoc.

Una policy è invece un formalismo per esprimere delle regole logiche del tipo:

$$\text{If } C_1\text{And...And } C_n\text{Then Set } A_k = V_k$$

Dove A_k è un attributo, V_k un valore e C_1 una condizione (del tipo $A_i < Valore$), per esempio, la policy dell'utente: "Quando sono in sala conferenze tutte le comunicazioni sul mio palmare devono avvenire in forma testuale" si traduce nella policy:

$$\text{If Location} = \text{'MConfRoom And Device} = \text{'PDA Then Set PreferredMedia} = \text{'text'}$$

Le regole inoltre possono essere etichettate e, per regole che specificano un valore per il medesimo attributo, espressioni del tipo $R_1 > R_2$ possono essere specificate per stabilire che R_1 ha una priorità maggiore rispetto ad R_2.

Una categorizzazione dei possibili conflitti che possono verificarsi e dei comportamenti desiderati comprende cinque casistiche:

1. *Conflitto tra valori espliciti per lo stesso attributo forniti da due differenti entità quando non nessuna policy è presente.* In questo caso la priorità fornita dalle direttive per la risoluzione dei conflitti determina quale valore prevale.

2. *Conflitto tra uno specifico valore per un attributo e una policy, data dalla stessa entità, da cui potrebbe derivare un nuovo valore per l'attributo.* Intuitivamente in questo caso la policy ha priorità maggiore rispetto all'attributo.

3. *Conflitto tra un valore esplicito per un attributo e una policy, data da un entità differente, da cui potrebbe derivare un nuovo valore per*

l'attributo. In questo caso per la risoluzione del conflitto è necessario ricorrere alle direttive: se l'entità che ha dichiarato la policy ha priorità minore rispetto a quella da cui arriva il valore esplicito, allora la policy può essere ignorata, altrimenti essa viene considerata e se da essa scaturisce un nuovo valore, questo prevale su quello esplicito.

4. *Conflitto tra due policy date da due differenti entità per uno specifico valore di un attributo*. Similmente al conflitto (3), la priorità è stabilità dalle direttive, in base alla priorità delle entità da cui proviene la policy.

5. *Conflitto tra due regole date dalla stessa entità per uno specifico valore di un attributo*. La priorità assegnata alla direttiva mediante l'espressione $R_1 > R_2$ stabilisce quale delle due regole considerare. Se nessuna priorità è stata impostata verrà utilizzato un ordinamento arbitrario.

Il modulo di aggregazione dati *Merge* è stato implementato in Java, utilizzando una libreria chiamata Mandrax per l'aggregazione dei profili XML.

Progetto del modulo *Profile Mediator Proxy* L'interfacciamento tra un qualsiasi servizio ed il *Context Provider* è delegato ad un'entità chiamata *Profile Mediator Proxy*, che è fondamentalmente un Proxy che intercetta le richieste HTTP provenienti dai Client, analizza gli headers per recuperare gli indirizzi dei profile manager, e richiede al *Context Provider* il relativo profilo utente. Una volta ricevuto quest'ultimo, lo inserisce negli headers HTTP della richiesta, inoltrandola poi alla logica applicativa.

Il modulo *Proxy Client* Lato Client vi è un Proxy locale che ha il compito di inserire negli headers HTTP della richiesta i dati relativi al profilo da recuperare, ovvero il nome utente e l'indirizzo dei *Profile Manager* da contattare.

Aggiornamento asincrono dei profili L'aggiornamento asincrono dei profili è affidato a dei trigger programmati dalla logica applicativa ed inoltrati

ai vari *Profile Manager*. Poiché il presente lavoro di tesi non ha riguardato questa funzione dell'architettura, la relativa descrizione viene omessa.

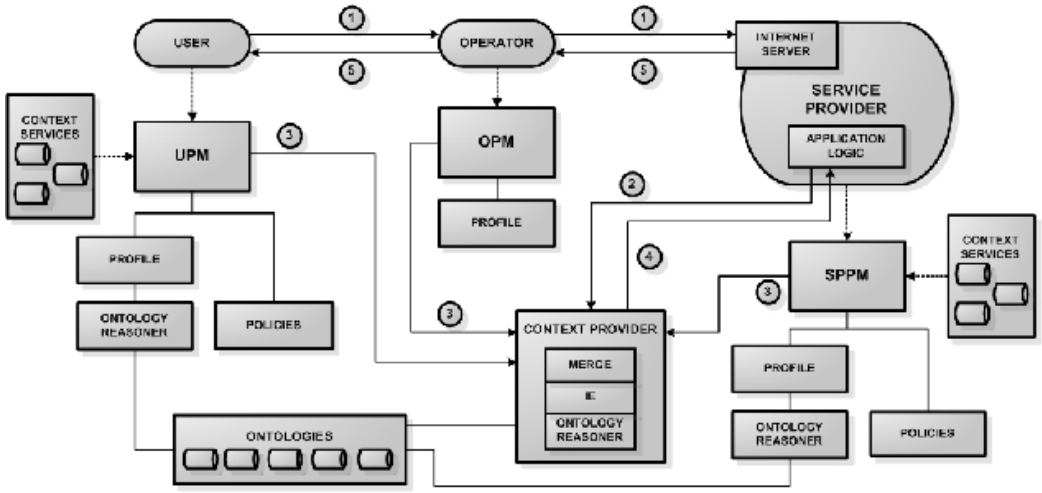


Figura 1.1: Flusso dei dati nell'architettura *CARE*

In figura 1.1 è mostrato il flusso dei dati durante una richiesta ad un ipotetico servizio. Come si può notare, ad ogni entità è associato un *Profile Manager*, il quale è responsabile della gestione del relativo profilo. Ogni volta che un utente fa una richiesta al *Service Provider* (fase 1) attraverso il suo dispositivo mobile, grazie al *Proxy Client* vengono aggiunti negli headers HTTP della richiesta gli URI necessari per contattare i *Profile Manager*.
Una volta ricevuta la richiesta, il *Profile Mediator Proxy* presente sul *Service Provider* spacchetta gli headers HTTP e si mette in contatto con il *Context Provider* (fase 2) per acquisire i dati di profilo necessari ad effettuare l'adattamento
Il Context Provider interroga quindi (agli URI specificati) i *Profile Manager* (fase 3) al fine di ottenere i dati di profilo e le policy dell'utente. I dati di profilo e le policy vengono quindi aggregati dal modulo *Merge* il quale ha il compito risolvere tutti i possibili conflitti che vengano ad evidenziarsi. I dati sono poi passati all'*Inference Engine* per la valutazione delle policy ed eventualmente possono essere utilizzati per effettuare dei ragionamenti ontologici.

Durante l'ultima fase (fase 4) il profilo viene inviato al *Profile Mediator Proxy*. Il PMP inserisce quindi il profilo aggregato negli headers della richiesta HTTP proveniente dal Client ed inoltra tale richiesta al *Service Provider*. A questo punto i dati di contesto sono pronti per essere utilizzati dalla logica applicativa che per adattare di conseguenza il servizio fornito al Client (fase 5).

1.2 Aree di intervento

Le prestazioni, in un'architettura che deve essere in grado di servire un numero alto di richieste contemporanee, sono un fattore critico. Perciò, basandosi sull'analisi della struttura dell'architettura e dei tempi di esecuzione di ogni singolo modulo si è cercato di individuare le possibili cause di lentezza. Si è così potuto constatare che i 200*millisecondi* necessari a servire una richiesta erano così distribuiti:

- circa 3*millisecondi* erano occupati dall'*Inference Engine*;
- circa 70*millisecondi* dal modulo *Merge*
- il restante tempo dalle query al database sui profile manager e dalla comunicazione fra moduli (*Profile Manager* e *Context Provider*, e fra *Context Provider* e *Profile Mediator Proxy*);

Si è pensato così di intervenire al fine di migliorare le ultime 2 voci . Per quanto riguarda l'aggregazione dei profili si è pensato di eseguire un'implementazione "ad-hoc", ovvero senza appoggiarsi su librerie XML esterne. Si ricordi che il precedenza l'aggregazione dati era realizzata sfruttando una libreria chiamata Mandrax.

Per migliorare invece la comunicazione tra i moduli, basandosi sull'articolo [6] si è pensato di testare altre tecnologie di comunicazione alternative ai Web Services. I Web Services, infatti, sono una tecnologia che permette di effettuare invocazione remota di metodi indipendentemente dalla piattaforma

e dal linguaggio utilizzato utilizzando un apposito protocollo basato su XML. Tuttavia questa grande flessibilità si paga in termini di efficienza, in particolare, come mostrato nell'articolo [7], vi è un grosso overhead computazionale dovuto alla serializzazione/deserializzazione della stringa XML, rispetto alla serializzazione binaria eseguita da RMI, oltre ad un sensibile overload della rete.

Bisogna sottolineare che la comunicazione tra i moduli della nostra architettura non necessita della versatilità dei Web Services, visto che è essenzialmente una comunicazione interna tra moduli programmati in linguaggio Java. Si è pensato perciò di cercare delle alternative a tale tecnologia, sperimentando altre tecnologie quali JNDI [12], RMI o, basandosi sull'articolo [8], le socket più la serializzazione binaria di oggetti Java.

Al fine di eseguire le suddette ottimizzazioni sono state perciò intraprese le seguenti azioni:

1. Progettazione di un formalismo per rappresentare i profili CC/PP, le policy e le direttive mediante oggetti Java, eliminando ogni tipo di serializzazione/deserializzazione XML;

2. Riscrittura degli algoritmi di aggregazione dei profili CC/PP e pesatura delle policy, che sfruttino la rappresentazione precedentemente definita, senza ricorrere a librerie esterne (Mandrax);

3. Scelta di un metodo di comunicazione tra *Profile Manager* e *Context Provider*, che sia più efficiente rispetto ai Web Services (RMI o socket più serializzazione binaria).

Capitolo 2

Reingegnerizzazione dei Moduli

In questo capitolo verrà esposto il lavoro di progettazione della libreria *Profiling* atta a rappresentare e gestire profili CC/PP e policy. Verrà poi mostrato come sono stati reingegnerizzati i *Profile Manager* ed in particolare verranno evidenziati i benefici relativi all'implementazione di una Pool di connessioni. Infine verrà mostrato come è stato riprogettato e reimplementato il modulo del *Context-Provider* per l'aggregazione dei profili CC/PP, ed i relativi algoritmi di aggregazione dei profili e pesatura delle policy.

Tutto il lavoro implementativo è stato svolto utilizzando il linguaggio Java, nella sua versione 1.4.2 e seguendo i principi della progettazione *Object Oriented* quali incapsulamento, ereditarietà, polimorfismo. Le qualità del software che più si è tenuto in considerazione sono correttezza, affidabilità, prestazioni ed evolvibilità, in un processo di sviluppo incrementale di tipo bottom-up.

2.1 La libreria Profiling

Figura 2.1: Diagramma del package *profiling*

Al fine di rappresentare esaustivamente il profilo di un utente senza ricorrere all'XML è stata creata una piccola libreria Java chiamata *profiling* contenente tutte le classi entità atte a rappresentare e gestire, mediante opportune procedure, profili di tipo CC/PP (Composite Capabilities/Preferences Profile) e regole policy) scritte con una sintassi di tipo Prolog. Questa libreria si compone di dodici classi divise in due sotto-package *ccppprofile* e *policy*, atti a rappresentare e gestire, rispettivamente profili CC/PP e regole.

Nel package principale, chiamato *profiling*, vi è un ulteriore Classe chiamata *ProfileAndPolicies*, avente lo scopo di unire, in una sola classe, sia le funzionalità di gestione del profilo sia quelle di gestione delle policy. Inoltre, incorporando in un solo oggetto sia il profilo CC/PP sia le policy, è possibile ottenere entrambe le entità tramite un'unica connessione ad un *Profile Manager*, risparmiando il tempo per la negoziazione del canale di comunicazione. Nel sotto-package *ccppprofile* vi sono le classi per la rappresentazione e gestione di un profilo CC/PP, mentre nel sotto-package *policy* vi sono le classi per la rappresentazione e gestione delle policy.

2.1.1 Modellazione del profilo CC/PP

In questa sezione sarà spiegato come è stata progettata ed implementata la libreria per la gestione di un profilo CC/PP all'interno della nostra architettura.
La specifica CC/PP (Composite Capabilities/Preferences Profile) è un raccomandazione del W3C che permette, utilizzando il linguaggio di markup

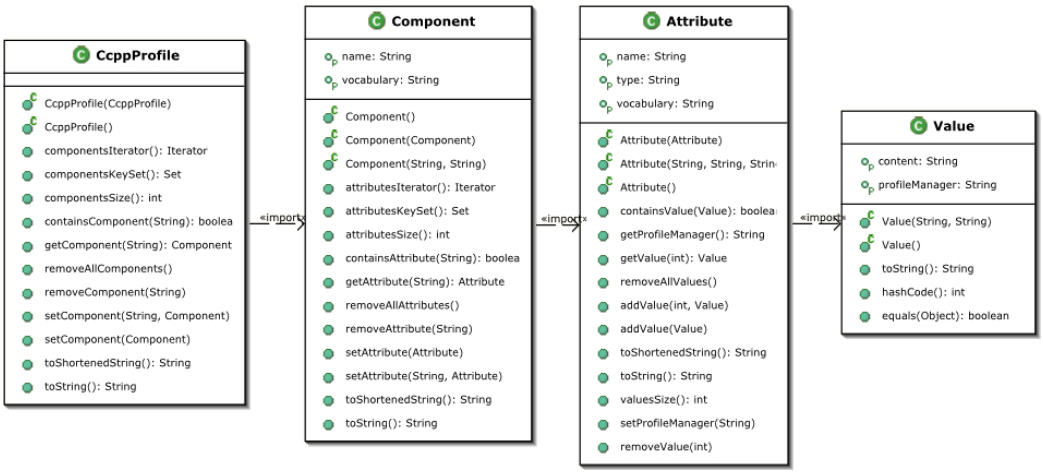


Figura 2.2: Diagramma del package *ccppprofile*

XML/RDF, di descrivere le capacità di un dispositivo mobile in un dato istante e le preferenze associate all'utente che lo utilizza; essa prevede una struttura gerarchica Componente/Attributo/Valore in cui ogni Profilo CC/PP contiene uno o più Componenti, ogni Componente è definito da un Vocabolario XML rappresentato da un URL e da un Nome e contiene uno o più Attributi identicamente definiti da un Vocabolario e da un Nome e cui è assegnato un Tipo che può essere:

- 'rdf:SIN' se l'attributo può contenere uno ed un solo valore;
- 'rdf:BAG' se l'attributo può contenere più valori non ordinati;
- 'rdf:SEQ' se l'attributo può contenere più valori ordinati sequenzialmente;

Ogni Attributo contiene infine uno o più componenti a seconda del tipo sopra citato.

Per modellare in linguaggio Java un profilo CC/PP è stato creato un sottopackage chiamato *ccppprofile* contenente le classi entità *CcppProfile*, *Component*, *Attribute* e *Value*, unite da una relazione di composizione. Tutte le classi di questo Package implementano l'interfaccia java.io.Serializable e rispettano il formalismo dei Java Bean, in modo che i relativi oggetti siano

facilmente serializzabili ed richiamabili tramite Web service. Secondo quanto precedentemente detto, un oggetto della classe *CcppProfile* è composto da uno o più oggetti della classe *Component*, un oggetto della classe Component si compone di uno o più oggetti della classe *Attribute*, mentre un *Attribute* può avere uno o più valori di tipo stringa di caratteri.

Una delle caratteristiche importanti per eventuali sviluppi futuri era la possibilità di conoscere il nome del *Profile Manager* da cui proviene un attributo. Questo è utile ad esempio dopo l'esecuzione dell'algoritmo di *Merge*, nel quale più profili provenienti da diversi *Profile Manager* vengono aggregati a formare un solo profilo. Nell'ambito di questa tesi si è pensato di permettere il tracciamento non solo del *Profile Manager* da cui proviene ogni attributo, ma anche del *Profile Manager* da cui proviene ogni singolo valore di ogni attributo. Questo perché, ad esempio, nell'algoritmo di *Merge* i valori degli attributi di tipo 'rdf:Seq' o 'rdf:Bag' provenienti dai diversi *Profile Manager* vengono accodati (secondo determinate direttive) in un nuovo attributo[1] ed è quindi interessante conoscere la provenienza di ciascuno di essi. Per questo è stata creata la classe *Value*, in modo da associare ad ogni valore un *Profile Manager*.

La struttura dati Java che è stata scelta per implementare la mappa di Componenti e di Attributi è la *java.util.HashMap*. La scelta è stata effettuata considerando che era necessario avere una struttura dati:

1. ridimensionabile dinamicamente a tempo di esecuzione, per permettere di inserire nel profilo un numero arbitrario ed indefinito di componenti e attributi;

2. indicizzabile tramite una stringa, per permettere di recuperare un componente o un attributo semplicemente effettuando una interrogazione contenente vocabolario e nome dello stesso;

3. accedibile sequenzialmente, per permettere di scorrere il contenuto di

[1]Per approfondimenti relativi all'algoritmo di *Merge* si veda la sezione 3.3.2

un profilo senza conoscerne gli indici (esempio per stamparlo a video in fase di sviluppo);

4. efficiente, dato che questo ero uno dei risultati che ci si era prefissato di ottenere.

Nella libreria J2SE1.4.2 la prima caratteristica è soddisfatta sia dalle classi che implementano l'interfaccia *List*, sia dalle classi che implementano l'interfaccia *Map*; la seconda caratteristica è soddisfatta solamente dalle classi che implementano l'interfaccia *Map*, in quanto le classi che implementano l'interfaccia *Collection* (*Vector*, *LinkedList*, *ArrayList*), come gli array, sono indicizzabili solamente tramite un intero e non tramite una stringa. La ricerca di una struttura dati appropriata si è ristretta quindi alle classi che implementano l'interfaccia *Map* (*TreeMap*, *HashTable* etc), tra le quali è stata scelta la classe *HashMap* perché garantisce un buon rapporto tra robustezza ed efficienza, assicurando tempi costanti per le principali operazioni quali *get*() e *put*(). Di contro questa struttura dati non è molto efficiente nella gestione dei dati, visto che la quantità di memoria allocata (misurata come numero di posizioni disponibili) è sempre superiore al numero di oggetti che contiene. Inoltre, ogni volta che viene raggiunta una certa percentuale di riempimento (chiamata *loadfactor*), la capacità della *HashMap* viene raddoppiata. Di conseguenza, una *HashMap* risulta avere problemi di performance nelle iterazioni non indicizzate, poiché queste includono anche posizioni vuote. Tuttavia è possibile una personalizzazione della capacità iniziale e del loadfactor e, se si analizzano le altre classi che implementano l'interfaccia *Map*, si scopre che non vi sono altre alternative egualmente valide:

1. *Hashtable*
   è una mappa di tipo *synchronized*, ovvero consente l'accesso ad una sola entità alla volta, utile nei contesti multithreading;

2. *TreeMap*
   è una mappa ordinata in ordine crescente, con tempo di accesso pari a $log(numeroElementi)$ per le operazioni *containsKey*, *get*, *put* e *remove*;

3. *WeakHashMap*
   è simile ad una *HashMap*, con la differenza che gli oggetti al suo interno non hanno caratteristiche di persistenza, ma vengono eliminati dal *Garbage Collector* di Java quando non sono più in uso da un certo periodo di tempo;

4. *IdentityHashMap*
   è una mappa che, per comparare chiavi e valori, adotta l'utilizzo della uguaglianza per riferimento piuttosto che per oggetto. Ad esempio, i metodi *containsKey*(o1) e *containsValue*(o1) verificano se due oggetti o1 e o2 referenziano una stessa area di memoria (o1==o2). La relazione tra i due oggetti è, come dice il nome, una identità.

La *HashTable* è stata scartata in quanto per le esigenze di questa tesi non era necessaria una struttura sincronizzata (la sincronizzazione richiede un overhead computazionale) dato che l'accesso in scrittura o lettura ai profili non avviene in multithreading. La *TreeMap* è stata scartata in quanto, sempre per le esigenze di questa tesi, non era necessario mantenere un ordinamento dei profili, soprattuto se questo comporta un tempo di lettura e scrittura che sale da costante (HashMap) a logaritmico. La *WeakHashMap* è stata scartata proprio per la caratteristica volatilità degli oggetti che sono memorizzati al suo interno. IdentityHashMap è stata scartata per via della reference-equality che mal si combina quasi con ogni tipo di oggetto. Ad esempio se si istanzia una stringa mediante la keyword *new String("Ciao")* e poi si confronta questa con la stringa "Ciao" mediante l'operatore di reference-equality si otterrà come risultato `false`. Se quindi si dovesse assegnare come chiave della *HashMap* un oggetto di tipo stringa costruito mediante la keyword new String(), non sarebbe più possibile recuperarlo, se non tramite la stessa istanza dell'oggetto precedentemente creato. Questo non avviene con le *HashMap* (ed in generale con tutte le altre mappe) in quanto queste classi verificano l'uguaglianza degli oggetti invocando il metodo .*equals*()). In particolare nel caso della classe *String* questo restituisce **true** se la sequenza di

caratteri di due stringhe da confrontare è la medesima, quindi non solamente se questi referenziano la stessa area di memoria.

La *HashMap* è stata quindi definitivamente scelta come struttura dati atta a contenere la mappa di Componenti e Attributi. Per quanto riguarda invece l'elenco di Valori associati ad ogni attributo, si è scelto di utilizzare un *ArrayList*. Questa scelta è motivata dal fatto che, nel caso dei valori, non era necessaria un indicizzazione mediante stringa, ma era sufficiente un'indicizzazione numerica. Si è scelta quindi la classe *ArrayList* che è un'implementazione ridimensionabile dinamicamente degli array, più performante di una *HashMap*. Infine, per la persistenza dei profili si è mantenuto l'uso di database *MySQL* gestiti dai *Profile Manager*.

Classe *CcppProfile* La classe *CcppProfile* è la struttura dati principale ed è in relazione di composizione con la classe *Component*. Essa fornisce una serie di metodi per l'accesso sequenziale o indicizzato in lettura o scrittura alla mappa (*HashMap*) di Componenti, indicizzati attraverso una stringa.

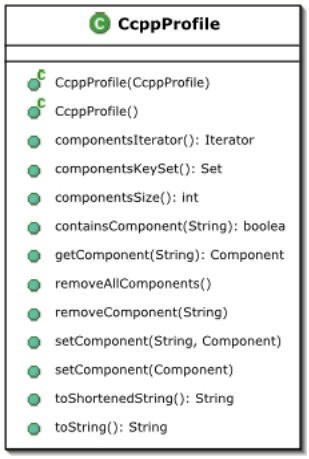


Figura 2.3: Diagramma della classe *CcppProfile*

L'implementazione sfrutta l'incapsulamento per nascondere la tipologia della mappa di componenti in modo da rendere possibile, se necessario, la modifica del tipo della mappa con un altra classe che implementi l'interfaccia

Map (in realtà non è neanche esplicito che si tratti di una mappa), rendendo tali modifiche invisibili agli utilizzatori della libreria. Questa classe ha due costruttori: uno che restituisce un profilo inizializzato e vuoto ed uno che restituisce un oggetto copia del profilo CC/PP che riceve come parametro formale. Inoltre implementa i seguenti metodi:

- `public Iterator componentsIterator()`
  restituisce un iteratore sui componenti;

- `public Set componentsKeySet()`
  restituisce un Set contenente tutte le chiavi (vocabolario e nome dei componenti). Esso è particolarmente utile per eseguire delle iterazioni simili a quelle eseguite da un ciclo *for*() su di un array, indicizzandole però, invece che con un intero, con una chiave di tipo stringa;

- `public int componentsSize()`
  restituisce il numero di componenti memorizzati nel profilo;

- `public boolean containsComponent(String resourceName)`
  per testare se nel profilo è memorizzato un componente il cui vocabolario e nome corrispondono a quello che si sta cercando;

- `public Component getComponent(String resourceName)`
  restituisce il componente, preso dal profilo, il cui "vocabolario#nome" corrisponde a quello della stringa in input.

- `public void removeAllComponents()`
  rimuove tutti i componenti dal profilo;

- `public void removeComponent(String resourceName)`
  rimuove il componente il cui "vocabolario#nome" corrisponde a quello della stringa in input;

- `public void setComponent(Component component)`
  memorizza un componente nel profilo. L'indice con cui tale componente

sarà memorizzato è generato automaticamente estraendo vocabolario e nome del componente dal componente passato in input;

- `public void setComponent(String resourceName, Component component)`
  memorizza, con un dato nome (resourceName), un componente nel profilo;

- `public String toShortenedString()`
  serializza una stringa (non XML) atta a rappresentare il profilo, per migliorarne la leggibilità viene eliminato il vocabolario di ogni componente e attributo. La serializzazione della stringa di caratteri viene eseguita in cascata: per ogni componente viene invocato il metodo toShortenedString, il quale invoca il metodo toShortenedString di ogni attributo, il quale invoca il metodo toString di ogni valore.

- `public String toString()`
  serializza una stringa (non XML) atta a rappresentare il profilo. La serializzazione della stringa di caratteri viene eseguita in cascata: per ogni componente viene invocato il metodo toString, il quale invoca il metodo toString di ogni attributo, il quale invoca il metodo toString di ogni valore.

Tramite questa classe è quindi possibile manipolare un profilo CC/PP, recuperando, memorizzando o rimuovendo Componenti, indicizzati in modo automatico o manuale, eseguendo delle iterazioni sulla mappa dei componenti o sulle chiavi di memorizzazione e serializzando una stringa atta a rappresentare il profilo nel modo che più si ritiene conveniente.

Classe *Component* La classe *Component* ha lo scopo di rappresentare un componente del profilo CC/PP ed è in relazione di composizione con la classe *Attribute*. Similmente alla classe *CcppProfile*, la classe *Component* fornisce una serie di metodi per accedere in modo sequenziale o indicizzato alla mappa (*HashMap*) di Attributi (*Attribute*), indicizzati attraverso una stringa.

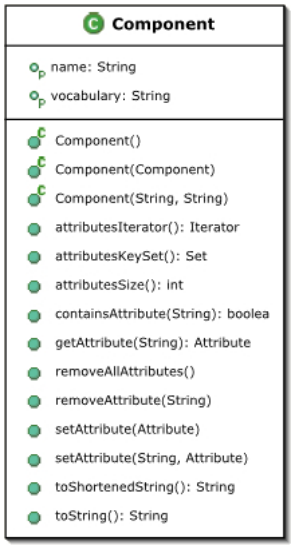


Figura 2.4: Diagramma della classe *Component*

Anche questa classe sfrutta l'incapsulamento per nascondere la tipologia della mappa di attributi in modo da permettere, se necessario, la modifica della struttura dati utilizzata per memorizzare gli attributi, senza ripercussioni sul codice esterno alla libreria. Questa classe ha tre costruttori: uno che restituisce un componente inizializzato e vuoto, uno che restituisce un componente copia del componente che riceve come parametro formale ed uno che crea un componente assegnandogli direttamente vocabolario e nome. Essa ha due proprietà che vengono accedute tramite gli opportuni metodi get() e set() e che rappresentano il vocabolario ed il nome del componente:

1. `name`
   rappresenta il nome del componente;

2. `vocabulary`
   rappresenta il vocabolario;

Questa classe implementa inoltre i seguenti metodi utili per la manipolazione del Componente:

- `public Iterator attributesIterator()`
  restituisce un Iteratore sugli attributi del componente;

- `public Set attributesKeySet()`
  restituisce un Set contenente tutte le chiavi (vocabolario#nome) degli attributi;

- `public int attributesSize()`
  restituisce il numero di attributi memorizzati nel componente;

- `public boolean containsAttribute(String resourceName)`
  controlla se nel componente è memorizzato un attributo la cui risorsa(vocabolario#nome) corrispondente a quella dell'attributo che si sta cercando;

- `public Attribute getAttribute(String resourceName)`
  restituisce un attributo, il cui vocabolario#nome corrisponde a quello della stringa in input;

- `public void removeAllAttributes()`
  rimuove tutti gli attributi dall'attuale componente;

- `public void removeAttribute(String resourceName)`
  rimuove l'attributo il cui vocabolario#nome corrisponde a quello della stringa in input;

- `public void setAttribute(Attribute attribute)`
  memorizza, nell'attuale componente, un attributo. L'indice con cui tale componente sarà memorizzato è generato automaticamente estraendo vocabolario e nome dall'attributo passato in input;

- `public void setAttribute(String resourceName, Attribute attribute)`
  memorizza, con un dato nome, un Attributo all'interno dell'attuale Componente;

- `public String toShortenedString()`
  serializza una stringa (non XML) atta a rappresentare il componente e tutti gli attributi contenuti. Da questa stringa, per migliorarne la

leggibilità, viene eliminato il vocabolario del componente e dei suoi attributi. La serializzazione della stringa di caratteri viene eseguita in cascata: per ogni attributo viene invocato il metodo toShortenedString, il quale a sua volta invoca il metodo toString di ogni valore.

- `public String toString()`
  serializza una stringa (non XML) atta a rappresentare il componente e tutti gli attributi contenuti. La serializzazione della stringa di caratteri viene eseguita in cascata: per ogni attributo viene invocato il metodo toString, il quale a sua volta invoca il metodo toString di ogni valore.

Tramite questa classe è quindi possibile manipolare un Componente, recuperando, memorizzando o eliminando Attributi, indicizzati in modo automatico o manuale, eseguendo delle iterazioni sulla mappa di Attributi o sulle chiavi di memorizzazione e serializzando, nel modo che più si ritiene conveniente, una stringa atta a rappresentare il componente.

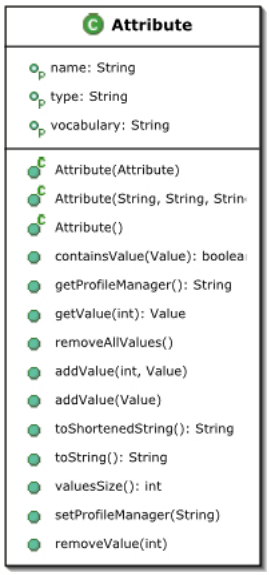


Figura 2.5: Diagramma della classe *Attribute*

Classe *Attribute* La classe *Attribute* serve a rappresentare un attributo contenuto all'interno di un componente. Questo attributo, secondo la

specifica CC/PP precedentemente citata, può essere di tre tipi: SIN, BAG, o SEQ. All'attributo saranno associati: nel primo caso (SIN) un singolo valore, nel secondo caso (BAG) un elenco non indicizzato di valori, nell'ultimo caso (SEQ) un elenco indicizzato di valori. Per gestire queste diverse tipologie in modo semplice e trasparente, si è pensato di utilizzare, come spiegato all'inizio di questa sezione, una *ArrayList* contenente: un singolo valore per gli attributi di tipi SIN; una sequenza di valori ordinati per gli attributi di tipo SEQ; una sequenza di valori indicizzati casualmente per gli attributi di tipo BAG. In particolare, per uniformarsi allo standard 'RDF:Seq'e 'RDF:Bag' l'indice di memorizzazione va da/a: 1¡=indice¡=valuesSize(). La classe è realizzata sfruttando l'incapsulamento e ha tre costruttori: uno che restituisce un attributo inizializzato e vuoto, uno che restituisce un attributo copia dell'attributo che riceve come parametro formale ed uno che crea un attributo assegnandogli direttamente vocabolario, nome e tipo. Essa ha tre proprietà che vengono accedute tramite gli opportuni metodi get() e set() e servono a rappresentare il vocabolario, il nome ed il tipo dell'attributo:

1. `name`
   rappresenta il nome del componente;

2. `vocabulary`
   rappresenta il vocabolario;

3. `type`
   rappresenta il tipo dell'attributo (SIN—BAG—SEQ);

Questa classe inoltre implementa i seguenti metodi utili per la manipolazione dell'attributo:

- `public boolean containsValue(Value valu)`
  controlla se l'attributo contiene un valore corrispondente a quello che si sta cercando. Questo metodo invoca il metodo contains() dell'*ArrayList* di valori, il quale a sua volta utilizza il metodo equals() della classe interessata (in questo caso *Value*) per verificarne la corrispondenza.

Secondo quanto definito in tale metodo due oggetti della classe *Value* sono uguali se e solo se il loro contenuto è uguale, indipendentemente dal profile manager di provenienza;

- `public String getProfileManager()`
  restituisce, se è definito, il *Profile Manager* da cui proviene un attributo. Questo metodo è utile per effettuare dei ragionamenti dopo l'operazione di *Merge*. Il *Profile Manager* di un attributo corrisponde al *Profile Manager* del valore in posizione 1;

- `public Value getValue(int index)`
  restituisce un valore, il cui indice corrisponde a quello del parametro index;

- `public void removeAllValues()`
  rimuove tutti i valori dall'attributo;

- `public void removeValue(int index)`
  rimuove il valore associato all'indice di memorizzazione passato come argomento;

- `public void setProfileManager(String profileManager)`
  imposta il profile manager per un attributo, impostando il *Profile Manager* per tutti i valori dell'attributo;

- `public void setValue(Value value)`
  assegna un valore all'attributo, adatto solo a gestire attributi di tipo SIN O BAG. Questo metodo assegna automaticamente un indice sequenziale al valore. Se l'attributo è di tipo SEQ è necessario invece utilizzare il metodo public void setValue(String position, Value value) per definire l'indice di memorizzazione del valore;

- `public void setValue(int index, Value value)`
  che assegna un valore all'attributo specificandone anche la posizione, utile per attributi di tipo SEQ;

- `public String toShortenedString()`
  serializza una stringa (non XML) atta a rappresentare l'attributo. Da questa stringa, per migliorarne la leggibilità viene eliminato il vocabolario dell'attributo. La serializzazione della stringa di caratteri viene eseguita in cascata invocando il metodo toString di ogni valore;

- `public String toString()`
  serializza una stringa (non XML) atta a rappresentare l'attributo. La serializzazione della stringa di caratteri viene eseguita in cascata invocando il metodo toString di ogni valore;

- `public int valuesSize()`
  restituisce il numero di valori associati all'attributo.

Tramite questa classe è quindi possibile manipolare un Attributo, recuperando o memorizzando Valori, indicizzati in modo automatico o manuale e serializzando, nel modo che più si ritiene conveniente, una stringa atta a rappresentare l'attributo.

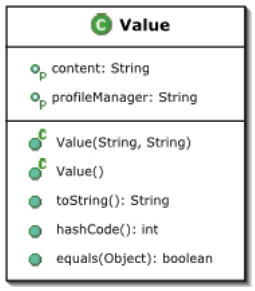


Figura 2.6: Diagramma della classe *Value*

Classe *Value* Come anticipato all'inizio del capitolo, il *Context Provider* deve memorizzare e recuperare il nome del *Profile Manager* da cui proviene ogni singolo Valore del profilo finale. Si è resa perciò necessaria la creazione di una classe *Value* con cui rappresentare sia il valore di un attributo sia il *Profile Manager* di provenienza. Questa classe ha due costruttori: uno

che restituisce un oggetto della classe *Value* inizializzato e vuoto, e uno che crea un oggetto value assegnandogli direttamente valore e *Profile Manager* di provenienza. Essa ha due proprietà che vengono accedute tramite gli appositi metodi get() e set():

1. `content:String`

2. `profileManager:String`

La prima proprietà memorizza il valore vero e proprio. Nella seconda proprietà viene memorizzato il *Profile Manager* da cui proviene il Valore. Questa classe effettua inoltre l'overriding dei metodi equals() e hashCode() ereditati dalla classe*Object*. La classe implementa inoltre i seguenti metodi:

- `public boolean equals(Object object)`
  sovrascrive il metodo equals() della classe *Object*. Secondo l'attuale implementazione due oggetti di classe *Value* sono identici se il valore della proprietà *content* è uguale;

- `public int hashCode()`
  sovrascrive il metodo hashCode() della classe *Object* in modo da rispettare la convenzione per cui due oggetti uguali (secondo il metodo .equals()) devono avere anche hashCode uguali;

2.1.2 Modellazione delle policies

Come precedentemente anticipato, una policy è un formalismo per esprimere un insieme di condizioni sul profilo dell'utente che, se soddisfatte, permettano di derivare un nuovo valore per un attributo. Nel nostro linguaggio una policy è espressa come una regola nella forma:

$$\text{If } C_1\text{And...And } C_n\text{Then Set } A_k = V_k$$

Dove A_k è un attributo, V_k un valore di profilo, mentre $C_1 C_n$ sono condizioni del tipo:

- $A_k = V_k$
- $A_k <> V_k$
- $A_k < V_k$
- $A_k > V_k$

Ogni condizione viene così espressa come un valore o un range di valori per un attributo di un determinato componente, che la rendono vera. L'azione da intraprendere nel caso in cui la condizione sia verificata è invece espressa come il valore da assegnare ad un attributo di un determinato componente.

Per gestire tramite calcolatore tali policies si è scelto di utilizzare una rappresentazione in formato stringa con una sintassi di tipo Prolog. Ogni policy è così composta da una testa e da un corpo. Nel nostro caso, la testa è formata da un unico atomo. Il corpo è formato dalla congiunzione di atomi (eventualmente negati). Ogni atomo è formato da un predicato e da un parametro, che può a sua volta essere una variabile o una costante. Seguendo questa sintassi, una policy come:

$$\text{If } A_k < V_k \text{And } A_h = V_h \text{Then Set } A_j = V_j$$

è quindi espressa mediante la sintassi:

$$\begin{gathered}\text{“Vocabolario\#Componente|Vocabolario\#Attributo}_j(\text{Valore}_j) \text{ :-} \\ \text{Vocabolario\#Componente|Vocabolario\#Attributo}_k(X), <(X, \text{Valore}_k), \\ \text{Vocabolario\#Componente|Vocabolario\#Attributo}_h(\text{Valore}_h) \text{ .”}\end{gathered}$$

Dove:

- La parte a sinistra è la testa della policy ed è composta da un unico atomo “Vocabolario#Componente|Vocabolario#Attributo$_j$(Valore$_j$)” composto di un predicato “Vocabolario#Componente|Vocabolario#Attributo$_j$” e di un parametro “Valore$_j$”.

- La parte a destra è invece il corpo della policy a sua volta composto da tre atomi:
  - "Vocabolario#Componente|Vocabolario#Attributo$_k(X)$" composto da un predicato "Vocabolario#Componente|Vocabolario#Attributo$_k$" e da una variabile "X";
  - "$< (X, \text{Valore}_k)$" composto da un predicato "<" e da un parametro "X,Valore$_k$" che a sua volta è una coppia: variabile "X", "costante Valore$_k$";
  - "Vocabolario#Componente|Vocabolario#Attributo$_h$(Valore$_h$)" composto da un predicato "Vocabolario#Componente|Vocabolario#Attributo$_h$" e da un parametro "(Valore$_h$)".

Una policy può non avere corpo ma non può non avere una testa. Una policy in cui sia presente solamente l'intestazione è rappresentata tramite la stringa:

"Vocabolario#Componente|Vocabolario#Attributo$_j$(Valore$_j$)."

e serve ad esprimere un valore per un attributo, senza nessuna precondizione. La rappresentazione delle policy in linguaggio Java è stata realizzata mediante il sottopackage *policy* comprendente cinque classi in relazione di composizione, più due classi per gestire le eccezioni:

- *Atom*, atta a rappresentare un atomo semplice composto da un un predicato e da un parametro. Nel nostro linguaggio questa classe viene utilizzata per rappresentare atomi il cui parametro è una variabile ed il cui predicato è un operatore di confronto come: $<, >, <=, >=, <>$;
- *CcppAtom*, eredita dalla classe *Atom* e serve a rappresentare un atomo che esprime una combinazione di Componente|Attributo e Valore di un profilo CC/PP. La particolarità di questo tipo di atomo è che il predicato è quindi composto da:
  Vocabolario#Componente|Vocabolario#Attributo;

- *PolicyPart*, atta a rappresentare una parte della policy, ovvero il corpo o l'intestazione, contenente più atomi;
- *Policy*, atta a rappresentare una policy nella sua interezza;
- *PoliciesList* atta a rappresentare e gestire un elenco di policies ordinate;
- *PolicyParseException*, avente lo scopo di modellare le possibili eccezioni in fase di parsing delle policy. Questa classe estende la java.lang.Exception;
- *IllegalPolicyPartException*, viene sollevata nel caso in cui si cerchi di assegnare come testa della *Policy* una *PolicyPart* contenente un numero di atomi diverso da uno. Questa classe estende la *java.lang.RuntimeException*.

La classe *PoliciesList* si compone di una o *n Policy*, la classe *Policy* è composta da due *PolicyPart*, mentre la classe *PolicyPart* si compone di uno o *n Atom* o *CcppAtom*; la struttura dati scelta per memorizzare l'elenco delle policies e degli atomi che le formano è l'ArrayList, per lo stesso motivo per cui era stata scelta per gestire l'elenco di valori di un attributo, ovvero per il fatto che è necessaria una semplice indicizzazione numerica su di una struttura dati dinamicamente ridimensionabile a tempo di esecuzione. La serializzazione e deserializzazione delle policy viene eseguita in cascata: le classi Atom e CcppAtom hanno il compito di serializzare e deserializzare un atomo, la classe *PolicyPart* utilizza le classi *Atom* e *CcppAtom* per serializzare e deserializzare tutti gli atomi di cui si compone, la classe *Policy* sfrutta la classe *PolicyPart* per serializzare e deserializzare l'intestazione ed il corpo della policy, mentre la classe *PoliciesList* usa la classe *Policy* per serializzare e deserializzare un elenco di policies.

La classe *IllegalPolicyPartException* estende la *java.lang.RuntimeException* (utilizzata per modellare eccezioni che possono non essere catturate, come ad esempio la *NullPointerException*) invece della *java.lang.Exception*. Questo perché si è pensato che il fatto che l'intestazione della policy debba avere uno ed un solo atomo sia un prerequisito per lavorare con le policies, così come il

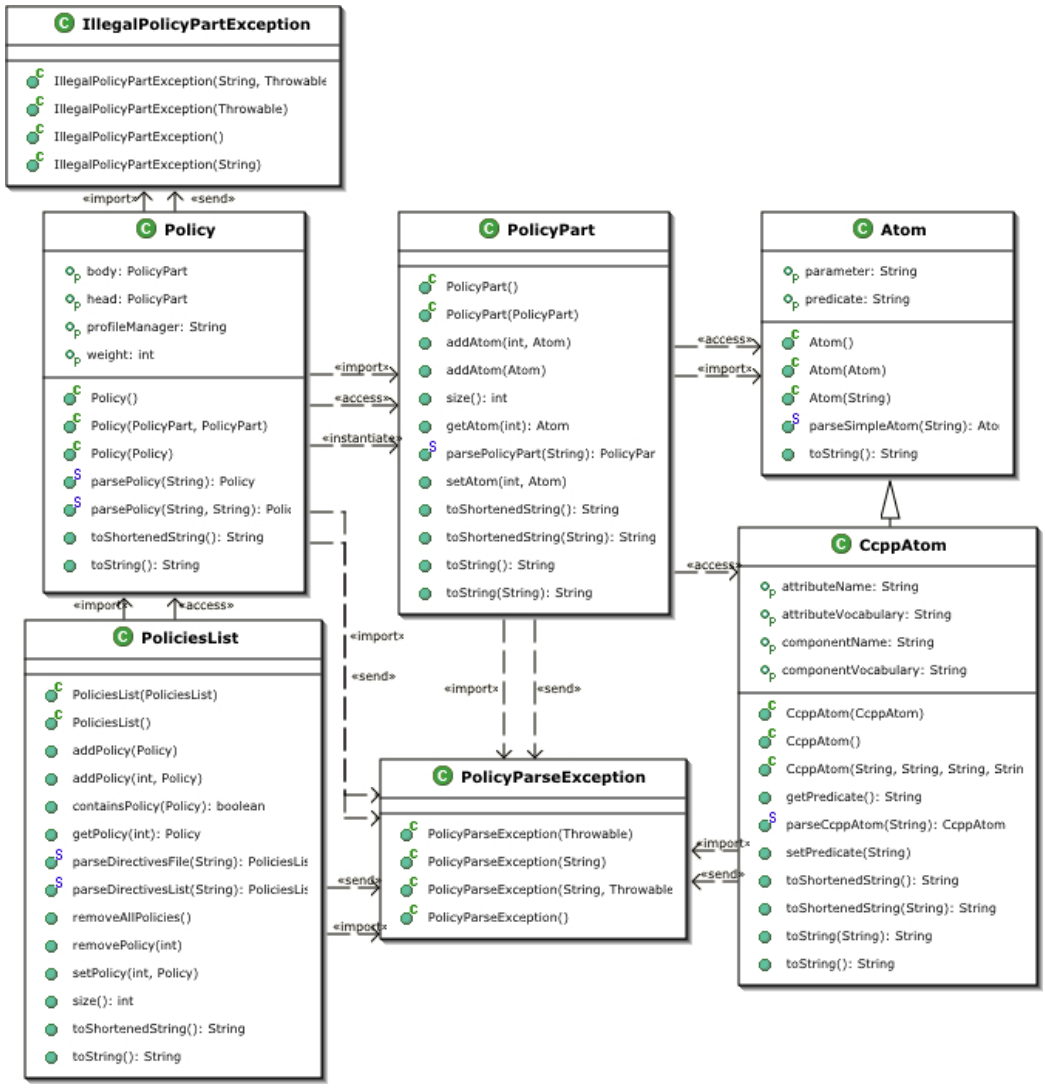


Figura 2.7: Diagramma del package *policy*

fatto che non si può invocare un metodo su un oggetto nullo sia un prerequisito generico. Perciò si è scelto di estendere la *java.lang.RuntimeException* che non obbliga gli utilizzatori della libreria a dover circondare le assegnazioni di una *PolicyPart* come testa della *Policy* con blocchi try{}catch{}. Il fatto che questa classe estenda la j*ava.lang.RuntimeException* che non necessita di essere catturata è il motivo per cui non è stato possibile modellare tutte le eccezioni del package policies mediante una sola eccezione.

Per garantire la persistenza delle policy è inoltre stata inserita nelle Basi di Dati gestite dai *Profile Manager* una tabella chiamata *Policy* avente i seguenti campi:

- *IDP*, che è un indice numerico che funge da chiave primaria della tabella;
- *IDU*, che è l'id dell'utente cui appartiene la policy;
- *policy*, contenente la stringa di testo che rappresenta la Policy;
- *weight*, in cui è memorizzato il peso della policy.

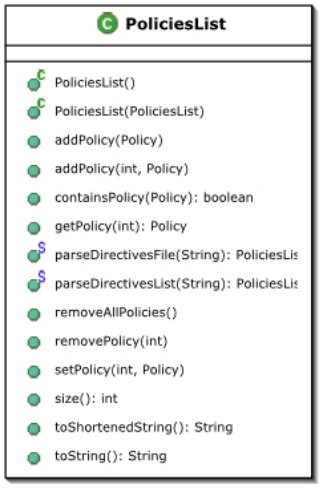


Figura 2.8: Diagramma della classe *PoliciesList*

Classe *PoliciesList* La classe *PoliciesList* ha il compito di rappresentare e gestire un elenco di policies modellate tramite la classe *Policy*. Essa implementa quindi i metodi per l'accesso sequenziale o indicizzato in lettura o scrittura all'elenco di policies, indicizzate numericamente. L'implementazione sfrutta l'incapsulamento per nascondere la classe dell'elenco di componenti in modo da rendere possibile, se necessario, la modifica del tipo di struttura dati utilizzata per gestire l'elenco di policies, rendendo tali modifiche invisibili agli utilizzatori della libreria. Questa classe ha due costruttori: uno che restituisce un profilo inizializzato e vuoto ed uno che restituisce un oggetto copia del profilo CC/PP che riceve come parametro formale; inoltre implementa i seguenti metodi:

- `public void addPolicy(Policy policy)`
  aggiunge una policy in fondo alla lista delle policies;
- `public void addPolicy(int index, Policy policy)`
  aggiunge una policy in una determinata posizione scalando a destra (aggiungendo 1 all'indice) la policy correntemente in quella posizione;
- `public boolean containsPolicy(Policy policy)`
  verifica se la lista contiene una policy memorizzata con un determinato indice;
- `public Policy getPolicy(int index)`
  restituisce una policy memorizzata con un determinato indice;
- `public static PoliciesList parseDirectivesFile(String fileName) throws PolicyParseException`
  per processare un file di testo contenente policies;
- `public static PoliciesList parseDirectivesList(String directivesList) throws PolicyParseException`
  per processare una stringa di testo contenente un elenco di policies, ogni policy deve stare su una riga differente;

- `public void removeAllPolicies()`
  elimina tutte le policies dal profilo;
- `public void removePolicy(int index)`
  rimuove la policy memorizzata con un determinato indice;
- `public void setPolicy(int index, Policy policy)`
  sovrascrive la policy il cui indice corrisponde a quello del parametro index;
- `public int size()`
  restituisce il numero di policies presenti nella lista;
- `public String toShortenedString()`
  serializza una stringa atta a rappresentare una lista di policies, da questa stringa, per migliorarne la leggibilità, viene eliminato il vocabolario di ogni componente e attributo;
- `public String toString()`
  serializza una stringa atta a rappresentare una lista di policies.

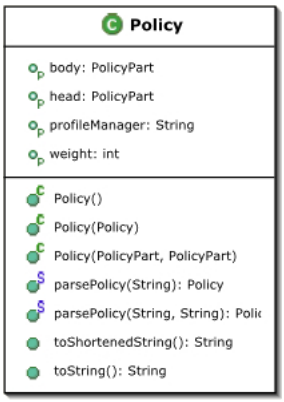


Figura 2.9: Diagramma della classe *Policy*

Classe *Policy* La classe *Policy*, in composizione con la classe *PolicyPart*, implementa tutto ciò che è necessario alla gestione di una policy. In particolare questa classe implementa i metodi per leggere e scrivere l'intestazione,

il corpo ed il peso di una policy, nonché i metodi per processare una stringa convertendola in un oggetto di tipo policy o serializzare una rappresentazione in formato stringa della policy stessa. Questa classe è realizzata sfruttando l'incapsulamento ed ha tre costruttori: uno che restituisce un oggetto della classe *Policy* inizializzato e vuoto, uno che restituisce una policy copia della policy che riceve come parametro formale ed uno che crea un oggetto della classe *Policy* assegnandogli direttamente intestazione e corpo. Essa ha quattro proprietà che vengono accedute tramite gli opportuni metodi get e set:

- `body:PolicyPart`
  rappresenta il corpo della policy;

- `head:PolicyPart`
  rappresenta la testa della policy. Il metodo setter di questa proprietà solleva una *IllegalPolicyPartException* nel caso in cui si tenti di assegnare come testa della policy una *PolicyPart* con un numero di atomi diverso da uno. Tuttavia, siccome l'eccezione sopracitata estende la classe *java.lang.RuntimeException* non è necessario circondare l'invocazione di questo metodo da un blocco trycatch;

- `profileManager:String`
  proprietà in cui è possibile memorizzare il *Profile Manager* da cui proviene una policy;

- `weight:int`
  rappresenta il peso eventualmente associato alla policy.

Questa classe implementa inoltre i seguenti metodi utili per il processing delle policy:

- `public String toShortenedString()`
  serializza una stringa, atta a rappresentare la Policy: in questo caso la stringa viene accorciata togliendo i vocabolari e lasciando solo il nome

dei componenti e attributi. Questa stringa non può però più essere processata dal metodo parsePolicy;

- `public String toString()`
  serializza una stringa, atta a rappresentare la Policy. Questa stringa può a sua volta essere data in input al metodo parsePolicy per riavere un oggetto Policy equivalente;

- `public static Policy parsePolicy(String policyString, String policyId)`
  `throws PolicyParseException`
  processa una policy in formato stringa restituendo un nuovo oggetto della classe *Policy*. Questo metodo solleva l'eccezione *PolicyParseException* di modo che venga correttamente catturata e gestita nel caso in cui la stringa atta a rappresentare la policy non sia correttamente formattata;

- `public static Policy parsePolicy(String policyString)`
  `throws PolicyParseException`
  processa una policy in formato stringa restituendo un nuovo oggetto della classe *Policy*. Anche questo metodo solleva l'eccezione *PolicyParseException* (in modo che venga correttamente catturata e gestita) nel caso in cui la stringa atta a rappresentare la policy non sia correttamente formattata. Questo metodo si occupa in particolare di verificare che la policy sia composta di due parti (corpo ed intestazione) separati dal simbolo corretto definito in una costante di classe.

Classe *PolicyPart* La classe *PolicyPart* ha il compito di gestire una singola sottoparte della policy (il corpo o l'intestazione), componendosi di uno o più oggetti della classe *Atom* o *CcppAtom*. Essa è realizzata sfruttando l'incapsulamento ed ha due costruttori: uno che restituisce un oggetto di tipo PolicyPart inizializzato e vuoto ed uno che restituisce una copia della *PolicyPart* che riceve come parametro formale. Essa implementa inoltre i

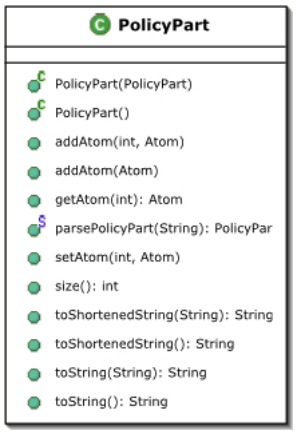


Figura 2.10: Diagramma della classe *PolicyPart*

seguenti metodi:

- `public void addAtom(Atom atom)`
  aggiunge un *Atom* infondo alla lista di atomi della *PolicyPart*;

- `public void addAtom(int index, Atom atom)`
  aggiunge, nella posizione specificata dall'indice, un *Atom* alla PolicyPart. L'atomo correntemente in quella posizione sarà scalato a destra di un posto;

- `public Atom geAtom(int index)`
  restituisce l'atomo specificato dall'indice;

- `public static PolicyPart parsePolicyPart(String policyPartString) throws PolicyParseException`
  processa una stringa restituendo un oggetto della classe *PolicyPart*. Solleva l'eccezione *PolicyParseException* (in modo che venga correttamente catturata e gestita) nel caso in cui qualcosa vada storto durante il parsing della policy. Questo metodo in particolare si occupa di verificare se gli atomi di cui è composta la stringa sono semplici (gli atomi semplici iniziano con una delle stringhe definite in una costante

di classe)o *CcppAtom* (i CcppAtom hanno come predicato il vocabolario#nome di un Componente—Attributo), processarli con i metodi della classe corretta ed infine memorizzarli;

- `public void setAtom(int index, Atom atom)`
  sovrascrive l'atomo il cui indice corrisponde a quello del parametro index;

- `public int size()`
  restituisce il numero di atomi presenti nella *PolicyPart*;

- `public String toShortenedString()`
  serializza una stringa con una sintassi di tipo Prolog, atta a rappresentare la *PolicyPart*. In questo caso la stringa viene accorciata togliendo i vocabolari e lasciando solo il nome dei componenti e attributi. Questa stringa non può però più essere processata dal metodo parsePolicyPart;

- `public String toShortenedString(String weight)`
  serializza una stringa con una sintassi di tipo Prolog, atta a rappresentare la *PolicyPart*, specificandone anche il peso. In questo caso la stringa viene accorciata togliendo i vocabolari e lasciando solo il nome dei componenti e attributi. Questa stringa non può però più essere processata dal metodo parsePolicyPart;

- `public String toString()`
  serializza una stringa con una sintassi di tipo Prolog, atta a rappresentare la *Policy Part*;

- `public String toString(String weight)`
  serializza una stringa con una sintassi di tipo Prolog, atta a rappresentare la *Policy Part*, specificandone anche il peso.

Classe *Atom* Un oggetto della classe *Atom* rappresenta un generico atomo composto da un predicato, il cui formato non è specificato e da un parametro.

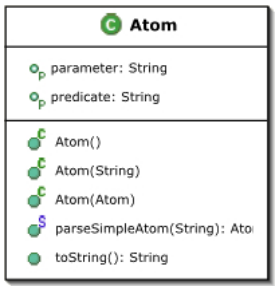


Figura 2.11: Diagramma della classe *Atom*

Nella nostra architettura viene utilizzata la classe *Atom* per rappresentare tutti quegli atomi il cui predicato inizia con un simbolo di comparazione (¡, ¿, ¿=, ¡=, ¡¿), ma essa può essere utilizzata per rappresentare qualsiasi tipo di atomi. Essa ha tre costruttori: uno che restituisce un oggetto della classe *Atom* inizializzato e vuoto, uno che restituisce una copia dell'oggetto della classe *Atom* che riceve come parametro formale ed uno che crea un atomo assegnandogli direttamente predicato e parametro. Essa ha due proprietà che vengono accedute tramite gli opportuni metodi get e set:

- `predicate:String`
  rappresenta il predicato dell'atomo;
- `parameter:String`
  rappresenta il parametro dell'atomo.

Questa classe implementa inoltre i seguenti metodi per il processing dell'atomo:

- `public String toString()`
  serializza una stringa atta a rappresentare l'atomo;
- `public static Atom parseAtom(String simpleAtomString)`
  processa una stringa e restituisce un nuovo oggetto della classe *Atom*.

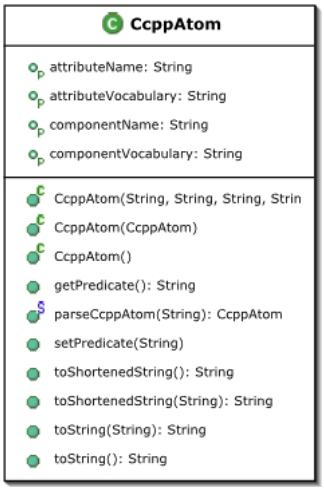


Figura 2.12: Diagramma della classe *CcppAtom*

Classe *CcppAtom* Questa classe eredita dalla classe *Atom* e si occupa di gestire un atomo che esprime una combinazione di Componente|Attributo e Valore di un profilo CC/PP. La particolarità di questo tipo di atomo è che il predicato deve esser composto da:
Vocabolario#Componente|Vocabolario#Attributo. Questa classe ha tre costruttori: uno che restituisce un oggetto della classe *CcppAtom* inizializzato e vuoto, uno che restituisce una copia del *CcppAtom* che riceve come parametro formale ed uno che crea un oggetto della classe *CcppAtom* assegnandogli direttamente vocabolario e nome al componente e all'attributo cui si riferisce. Essa ha quattro proprietà che servono a rappresentare il predicato dell'atomo e vengono accedute attraverso gli opportuni metodi get e set.

- `attributeName:String`
  rappresenta il nome dell'attributo;
- `attributeVocabulary:String`
  rappresenta il vocabolario dell'attributo;
- `componentName:String`
  rappresenta il nome del componente;

- `componentVocabulary:String`
  rappresenta il vocabolario del componente.

Questa classe implementa inoltre i seguenti metodi:

- `public String getPredicate()`
  override del metodo getPredicate ereditato dalla classe *Atom*. Restituisce il predicato, componendo nomi e vocabolari di Attributo e Componente CC/PP.

- `public static CcppAtom parseCcppAtom (String ccppAtomString) throws PolicyParseException`
  processa una stringa restituendo un oggetto della classe *CcppAtom* e solleva l'eccezione *PolicyParseException* (in modo che venga correttamente catturata e gestita) nel caso in cui la stringa atta a rappresentare l'atomo non sia correttamente formattata. In particolare questo metodo si occupa di controllare che:

  – Vocabolario e nome del componente o attributo siano separati dal simbolo "#";

  – Componente e attributo siano separati dal simbolo "|";

  – Componente|Attributo siano seguiti dal valore del predicato tra parentesi tonde "()".

  La sintassi accettata è quindi:
  "vocabolario#nomeComponente|vocabolario#nomeAttributo(Valore)".
  Per effettuare questa operazione viene invocato un metodo privato chiamato parsePredicate(String predicate) che si occupa di processare il predicato dell'atomo.

- `public void setPredicate(String predicate)`
  override del metodo setPredicate, ereditato dalla classe *Atom*, che processa la stringa in input e assegna un valore alle proprietà atte a rappresentare il predicato CC/PP. Per effettuare questa operazione viene

invocato un metodo privato chiamato parsePredicate(String predicate) che si occupa di processare il predicato passatogli come parametro formale. Sarà poi il metodo setPredicate() a gestire eventuali *PolicyParseException*.

- `public String toShortenedString()`
  serializza una stringa, atta a rappresentare un atomo. In questo caso la stringa viene accorciata togliendo i vocabolari e lasciando solo il nome di componente e attributo.

- `public String toShortenedString(String weight)`
  serializza una stringa, atta a rappresentare un atomo, specificandone anche il peso. In questo caso la stringa viene accorciata togliendo i vocabolari e lasciando solo il nome di componente e attributo.

- `public String toString()`
  serializza una stringa atta a rappresentare un atomo.

- `public String toString(String weight)`
  serializza una stringa atta a rappresentare un atomo, specificandone anche il peso.

Classe *ProfileAndPolicies* Questa è l'unica classe del package *profiling*, essa è stata pensata per rappresentare completamente il profilo di un utente, includendo sia il profilo CC/PP che le policies. Ciò permette inoltre la ricezione di entrambe le entità stesso mediante una sola transazione. Questa classe ha due costruttori: uno che restituisce un oggetto della classe *ProfileAndPolicies* inizializzato e vuoto ed uno che restituisce una copia dell'oggetto ProfileAndPolicies che riceve come parametro formale. Essa si compone quindi di un profilo CC/PP rappresentato mediante la classe *CcppProfile* e di un elenco di policies rappresentate tramite la classe *PoliciesList*. Entrambe queste entità sono implementate come proprietà accedibili attraverso gli opportuni metodi get e set:

- `profile:CcppProfile`
  rappresenta il profilo CC/PP;

- `policies:PoliciesList`
  rappresenta l'elenco delle policies.

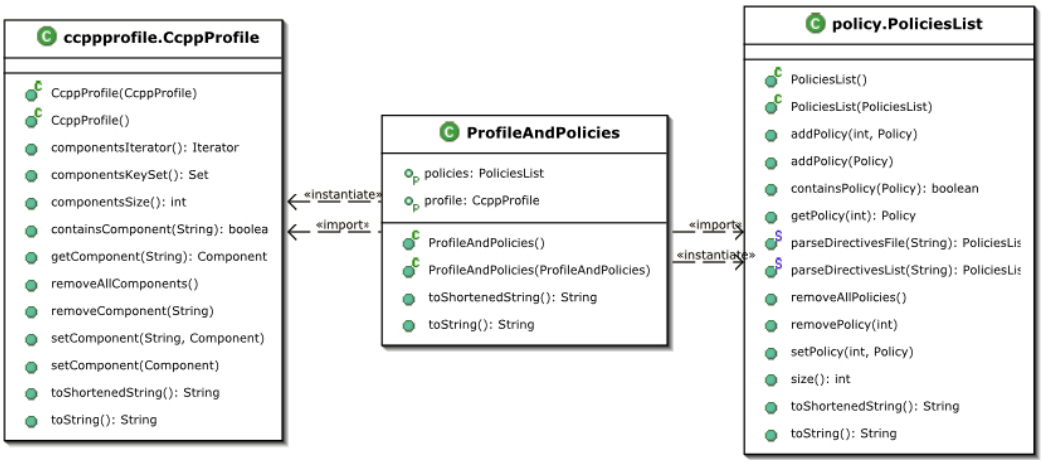


Figura 2.13: Diagramma del package *profiling*

Questa classe implementa inoltre i seguenti metodi:

- `public String toShortenedString()`
  serializza una stringa atta a rappresentare un profilo, comprensivo di policies. Da questa stringa, per migliorarne la leggibilità, viene eliminato il vocabolario di ogni componente e attributo.

- `public String toString()`
  serializza una stringa atta a rappresentare un profilo, comprensivo di policies.

2.2 Reingegnerizzazione dei *Profile Manager*

I *Profile Manager* sono le entità aventi come compito principale la gestione dei profili utente. Nell'ambito del lavoro di questa tesi sono essi sono stati riprogettati per diventare ognuno un'applicazione J2EE (Java 2 Enterprise Edition) programmata per componenti. Inoltre sono stati riprogettati e reimplementati i moduli di connessione al database e di composizione dei profili, nonché i moduli di comunicazione tra *Profile Manager* e *Context Provider*. In questa sezione ci occuperemo dei sottomoduli per la connessione al database e per la composizione dei profili, nonché della composizione dell'applicazione J2EE per la gestione dei *Profile Manager*, tralasciando come questi comunichino con il *Context Provider*. Per quanto riguarda la connessione al database si è abbandonato il vecchio metodo di connessione che imponeva ai *Profile Manager* di stabilire una nuova connessione ogni volta che era necessario effettuare un'interrogazione per recuperare un profilo. Tale metodo è stato abbandonato in quanto comportava un notevole overhead computazionale per l'autenticazione e l'instaurazione del canale di comunicazione. Ispirandosi all'implementazione di alcune applicazioni Web (ad esempio Apache Cocoon) è stata implementata invece una Pool di connessioni al Database che permette di risparmiare il tempo della negoziazione del canale di comunicazione, non chiudendo le connessioni al termine dell'interrogazione e rendendole disponibili per le successive eventuali interrogazioni.

2.2.1 Progettazione dell'applicazione J2EE

Al fine di facilitare la gestione dei moduli *Profile Manager* ognuno di essi è stato riprogettato come un'applicazione J2EE (Java 2 Enterprise Edition) [10]. La piattaforma J2EE è uno standard aperto per lo sviluppo di applicazioni Enterprise con il linguaggio Java comprendente diverse tecnologie quali ad esempio:

- Servizi di naming - JNDI (Java Naming and Directory Interface);
- Accesso a database - JDBC (Java DataBase Connectivity);
- Messaging asincrono - JMS (Java Messaging Service);
- Programmazione per componenti - EJB (Enterprise Java Beans);

In particolare, le tecnologie utilizzate per la realizzazione dell'applicazione J2EE atta a gestire un *Profile Manager* sono le seguenti:

1. POJO(Plain Old Java Objects). Qualsiasi classe convenzionale della piattaforma J2SE (Java Standard Edition), che non rientra in altre categorie, è una classe di tipo POJO.
2. JavaBean. È uno standard della piattaforma J2SE che permette di creare dei componenti (Bean) altamente riutilizzabili ed interoperabili. Esso definisce alcune convenzioni nella progettazione delle classi, tra cui la più usata è senz'altro quella che sancisce che ogni attributo di classe non debba essere acceduto direttamente, ma solo tramite degli opportuni metodi get<Attributo> e set<Attributo>. Seguendo tale convenzione è possibile pre-processare i valori da assegnare o leggere dall'attributo, mascherare la classe di appartenenza dell'attributo, nonché definire permessi di sola lettura o sola scrittura per il suddetto attributo. Tale attributo prende il nome di proprietà.
3. Servlet. È una tecnologia della piattaforma J2EE che permette di scrivere delle classi che implementino un interfaccia web e che siano richiamabili appunto mediante una richiesta HTTP. Una Servlet viene caricata ed inizializzata soltanto alla prima invocazione ed integra la gestione della concorrenza in modo da poter servire più richieste contemporaneamente. Unitamente alla tecnologia JSP (Java Server Pages) è possibile utilizzare le Servlet per sviluppare applicazioni web. Un'applicazione web in Java viene pubblicata in opportuni pacchetti *.war* contenenti le Servlet e le pagine JSP opportunamente collegate.

4. JNDI (Java Naming and Directory Interface) [12] è un interfaccia per l'accesso a sistemi di naming e directory quali ad esempio LDAP (Lightweight Directory Access Protocol) o DNS (Domain Name System). JNDI offre un livello di astrazione nei confronti dei sistemi di naming simile a quello offerto da JDBC nei confronti dei database relazionali, esso infatti richiede che un *provider* (o driver) implementi le funzionalità di accesso al sistema di naming. Il package JNDI viene fornito di default con alcuni service provider per connettersi ai principali sistemi di directory attualmente disponibili: LDAP, NIS, COS (CORBA Object Service), RMI Registry, File System.

5. EJB (Enterprise Java Beans). [11] È una tecnologia della piattaforma J2EE che permette di definire, similmente ai JavaBean, dei componenti (Bean) altamente riutilizzabili ed interoperabili. Un EJB non gode di vita propria ma, per essere eseguito, necessità di un EJB Container cui è affidata la gestione del suo ciclo di vita. Esistono tre categorie di Bean:

   - *Entity Bean*: analogamente alle classi entità, essi costituiscono la rappresentazione delle entità dei database;
   - *Session Bean*: analogamente alle classi di controllo, essi includono la logica del sistema e forniscono quindi determinati servizi o funzionalità;
   - *Message-Driven Bean*: servono a ricevere messaggi asincroni mediante la tecnologia JMS (Java Messaging Service);

   L'interazione tra un EJB ed il suo container può avvenire in tre modi:

   - *metodi callback*: ogni EJB implementa un'apposita interfaccia in cui sono definiti vari metodi callback che comunicano al Bean il verificarsi di determinati eventi come attivazione, persistenza e terminazione di una transazione. Nei *Session Bean*, in tale interfaccia è possibile inoltre definire dei metodi "business". Un meto-

do “business” è una funzionalità (o servizio) che si vuole fornire attraverso l'EJB;

- l'oggetto *EJBContext*: ogni EJB mantiene un riferimento al container mediante l'interfaccia EJBContext attraverso cui può richiedere informazioni sull'identità dei suoi Client o sullo stato di una transazione;
- *JNDI*: ad ogni EJB è associato un nome univoco, la mappatura del nome con il componente vero e proprio è affidata alla tecnologia JNDI. Attraverso JNDI ogni EJB ha automaticamente accesso ad uno speciale sistema di naming chiamato ENC (Environment Naming Context) gestito dall'EJB container. Come detto sopra JNDI necessita di un provider per ogni sistema di naming ed è quindi compito dell'EJB Container implementare e fornire il provider per l'accesso all'ENC. Attraverso ENC ogni Bean può accedere a diverse risorse quali, ad esempio, altri EJB o connessioni JDBC.

La tecnologia EJB tuttavia non è completamente standardizzata e alcune parti di essa, come la creazione automatica delle classi e delle interfacce necessarie al funzionamento, nonché la compilazione ed il deploy (pubblicazione) dei suddetti EJB dipendono dal container. Ciò significa che è il container EJB a doversi occupare di eseguire le suddette operazioni, quindi, ad esempio processo di sviluppo di un EJB per il container JBoss è diverso da quello per il container WebLogic della Sun.

Nell'ambito di questo lavoro di tesi le suddette tecnologie sono state combinate al fine di creare una applicazione J2EE composta da diversi componenti il più possibile disaccoppiati e coesi. Per raggiungere quindi tale fine e massimizzare la coesione ed il disaccoppiamento dei moduli, la suddetta applicazione è stata realizzata utilizzando la cosiddetta programmazione per componenti in standard J2EE. In particolare, ogni *Profile Manager* è ora una applicazione J2EE composta nel seguente modo:

- Tutte le classi entità e di controllo del *Profile Manager*, quali ad esempio la libreria *profiling*, le classi *ConnectionPool* e *ProfileUtils* e le classi per il controllo di server basati sulle Socket o su RMI (che saranno spiegate nella sezione 2.4), sono state archiviate in un "normale" archivio jar.

- Per ogni *Profile Manager* è stato creato un *Session Bean*, a sua volta contenente l'archivio jar sopracitato. Tale EJB implementa dei metodi "business" per il recupero del profilo associato ad un utente ed il controllo dei server, i quali a loro volta, per eseguire le funzionalità richieste istanziano le opportune classi del suddetto archivio. Ad ogni EJB, tramite JNDI, è associato un nome univoco (ad esempio "*UPM*") che è possibile utilizzare per ottenere un riferimento al componente.

- È stato creato un archivio contenente un'applicazione web associata al *Profile Manager*. Tale applicazione si compone di alcune pagine html e di alcune Servlet per controllare i server sopracitati o visualizzare un profilo CC/PP. Le Servlet utilizzano JNDI per ottenere dal container un riferimento all'EJB (mediante una ricerca per nome, come ad esempio "*UPM*"), dopodiché effettuano un'invocazione remota degli appropriati metodi 'business" dell'EJB, presentando infine il risultato opportunamente formattato.

Utilizzando questo stile di composizione dell'applicazione si è reso possibile disaccoppiare completamente la logica dei *Profile Manager* rispetto all'applicazione web. Infatti l'applicazione web non include nessuna classe o libreria di quelle che sarebbero state necessarie per realizzare le funzionalità che fornisce, se essa non fosse stata programmata utilizzando un approccio per componenti. Essa utilizza invece un riferimento remoto all'EJB ed è quindi completamente disaccoppiata rispetto alla logica applicativa. Diventa così possibile, per esempio, realizzare applicazioni che utilizzano componenti indipendentemente dalla locazione fisica di questi all'interno del file-system o che sono addirittura ubicati in container diversi su macchine diverse. Per esempio l'applicazione web per la gestione di un *Profile Manager* potrebbe

essere contenuta in un Servlet Container su di una macchina, mentre l'EJB contenente la logica del medesimo *Profile Manager* in un EJB container su di un'altra macchina. Ma anche all'interno del medesimo container è possibile notare come si possa fisicamente spostare l'EJB da una cartella all'altra del container continuando questo ad essere sempre accessibile dall'applicazione web senza che sia necessario cambiare nemmeno una riga di codice. In figura

2.2.2 Package *db*

In questo package sono incluse le classi per la connessione al database e la composizione del profilo CC/PP e delle policy associate ad un utente. In figura 2.14 è possibile vedere le relazioni che intercorrono tra le classi che lo compongono.

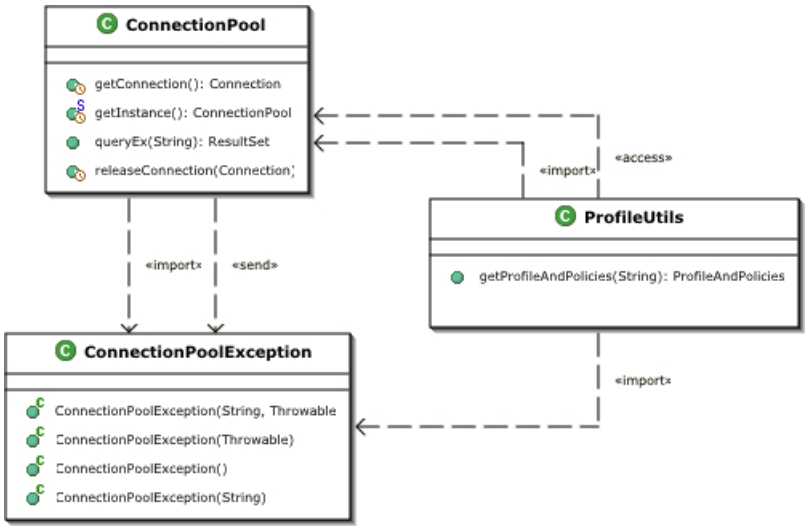


Figura 2.14: Diagramma del package *db*

Classe *ProfileUtils* Questa classe ha il compito di recuperare il profilo CC/PP e le policies associate ad un utente e di generare gli oggetti atti a rappresentarli. Per garantire la persistenza delle policies è stata inserita una tabella nel database di ogni *Profile Manager* cui, per evitare l'omonimia con

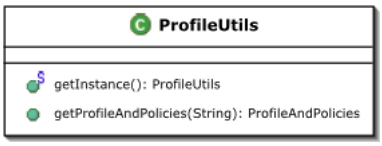


Figura 2.15: Diagramma della classe *ProfileUtils*

una tabella legacy (*policies*), è stato dato il nome al singolare *policy*. Tale tabella è composta da tre campi che non possono essere nulli:

- *IDP*, di tipo auto_increment, che è la chiave primaria della tabella contenente un id univoco per ogni policy;
- *idu*, contenente l'id dell'utente a cui è associata la policy;
- *policy*, contenente la policy vera e propria;
- *weight*, contenente il peso della policy.

La classe *ProfileUtils* rappresenta una sorta di interfaccia verso i database che può essere interrogata al fine recuperare quanto detto sopra. Questa classe è implementata seguendo il design pattern Singleton [9] e non ha quindi costruttore pubblico, per ottenerne un istanza è necessario utilizzare il metodo getInstance(). Essa implementa inoltre i seguenti metodi:

- `public static ProfileUtils getInstance()`
  restituisce una istanza condivisa della classe *ProfileUtils*;
- `public ProfileAndPolicies getProfileAndPolicies(String userName)`
  Questo metodo si occupa di comporre un oggetto contenente sia il profilo CC/PP, sia le policies associate ad un determinato utente. Esso è stato implementato in modo da eseguire due sole query e due soli cicli per riempire l'oggetto ProfileAndPolicies: una per recuperare un ResultSet per il profilo CC/PP ed una per recuperare le policies. Tale scelta è stata effettuata per minimizzare il numero di interrogazioni da effettuare per recuperare un profilo, soprattutto in relazione al fatto che inizialmente per ogni interrogazione richiedeva una nuova connessione

al database. L'interrogazione per recuperare il profilo CC/PP viene così effettuata mediante l'unione (Join) di cinque tabelle: *attributes*, *components*, *namespaces*, *profile*, *users*, ordinando i risultati in base all' *id* del componente, all'*id* dell'attributo e alla posizione di ogni valore, dove *id* del componente e *id* dell'attributo sono degli identificatori univoci.

L'interrogazione per recuperare tutte le policy associate ad un utente è molto semplice, dato che viene effettuata su un Join di due tabelle: policy e users. Per poter invece comporre il profilo CC/PP associato ad un utente mediante una sola interrogazione ed un solo ciclo senza sottocicli, l'algoritmo implementato è il seguente.

Algorithm 1 fillProfile

Ensure: Genera un oggetto della classe *CcppProfile* atto a rappresentare il profilo CC/PP e le policies associate ad un utente.

$ccppProfile \Leftarrow newCcppProfile$;
for all $resultSet_x \in ResultSets$ **do**
  $idComponent_x \Leftarrow resultSet_x.getIdComponent()$;
  if $idComponent_x.isChanged()$ **then**
    $component_x \Leftarrow newComponent(resultSet_x.getComponentData())$;
    $ccppProfile.setComponent(component_x)$;
  end if
  $idAttribute_x \Leftarrow resultSet_x.getIdAttribute()$;
  if $idAttribute_x.isChanged() || idComponent_x.isChanged()$ **then**
    $attribute_x \Leftarrow newAttribute(resultSet_x.getAttributeData())$;
    $component_x.setAttribute(attribute_x)$;
  end if
  $value_x \Leftarrow newValue(resultSet_x.getValueData())$;
  $attribute_x.addValue(value_x)$;
end for
return $ccppProfile$;

Notazione dell'algoritmo 1

Molti dei metodi citati (come $resultSet_x.getComponentData()$) sono delle astrazioni per facilitare la comprensione.

- $ccppProfile$ è un oggetto della classe *CcppProfile*;
- *ResultSets* è l'insieme delle tuple $resultSet_x$ ottenute in risposta all'interrogazione;
- $resultSet_x$ è una tupla contenente tutti i valori dei campi selezionati dall'interrogazione;
- $idComponent_x$ è un identificatore univoco per ogni componente;
- $resultSet_x.getIdComponent()$ recupera dalla tupla e restituisce l'id del componente;
- $idComponent_x.isChanged()$ restituisce `true` se, rispetto all'iterazione precedente, l'identificatore del componente è cambiato;
- $component_x$ è un oggetto della classe *Component* che verrà memorizzato nel profilo $ccppProfile$;
- $resultSet_x.getComponentData()$ recupera dalla tupla e restituisce i dati relativi al componente attuale, come nome e vocabolario;
- $ccppProfile.setComponent(component_x)$ aggiunge il componente $component_x$ al profilo $ccppProfile$;
- $idAttribute_x$ è un identificatore univoco per ogni attributo;
- $attribute_x$ è un oggetto della classe *Attribute* atto a rappresentare un attributo contenuto nel componente $component_x$;
- $resultSet_x.getIdAttribute()$ recupera dalla tupla e restituisce l'id dell'attributo;

- $idAttribute_x.isChanged()$ restituisce `true` se, rispetto all'iterazione precedente, l'identificatore dell'attributo è cambiato;
- $resultSet_x.getAttributeData()$ recupera dalla tupla e restituisce i dati relativi all'attributo attuale, come nome e vocabolario;
- $component_x.setAttribute(attribute_x)$ aggiunge l'attributo $attribute_x$ al componente $component_x$;
- $value_x$ è un oggetto della classe *Value* atto a rappresentare un valore contenuto nell'attributo $attribute_x$;
- $resultSet_x.getValueData()$ recupera dalla tupla e restituisce un valore dell'attributo $attribute_x$;
- $attribute_x.addValue(value_x)$ aggiunge all'attributo $attribute_x$ il valore $value_x$.

2.2.3 Pool di Connessioni

Una "pool di connessioni" è un'insieme condiviso di connessioni verso un'entità, disponibili per essere utilizzate a qualche scopo e poi rilasciate quando non se ne fa più uso. Condividendo un numero di connessioni che varia in base al numero di richieste da servire, tale pool permette, ogni volta che è necessario effettuare qualche tipo di comunicazione, di evitare la procedura di connessione, sfruttando le connessioni già disponibili. Tecnicamente essa è sostanzialmente una coda in cui vengono stipate e, quando necessario recuperate (per il tempo necessario), le connessioni disponibili. L'utilizzo di una pool di connessioni al database è stato pensato in quanto l'architettura oggetto di questa tesi vorrebbe poter gestire un elevato numero di Client contemporaneamente e questo (senza adeguate tecniche di caching che non sono ancora state implementate) significa un elevato numero di connessioni al database. Ciò rende inadeguato il canonico metodo per cui si effettua una nuova connessione ogni qual volta sia necessario effettuare una query. Perciò,

prendendo spunto da quanto fatto da alcune applicazioni web (ad esempio Apache Cocoon) e con lo scopo di realizzare un meccanismo di connessione più efficiente, si è pensato di inserire anche in questa architettura una prima implementazione di una "connection-pool".
Nonostante esistessero diverse librerie OpenSource pronte all'uso, per personale curiosità e voglia di sperimentare, essa è stata implementata manualmente e, come accennato in precedenza è sostanzialmente una coda che funziona nel seguente modo:

- ogni volta che è necessario eseguire una query viene effettuato un controllo per vedere se (nella coda delle connessioni libere) è disponibile una connessione su cui poterla eseguire;
- in caso contrario (al primo tentativo di connessione o quando tutte le connessioni disponibili sono occupate) una nuova connessione viene creata, assegnata alla query per il tempo necessario all'esecuzione della query stessa e liberata (aggiunta alla coda delle connessioni disponibili) quando la query è stata completata;
- in caso affermativo la prima connessione disponibile viene tolta dalla coda, assegnata alla query per il tempo necessario all'esecuzione della query stessa e liberata (ri-aggiunta alla coda delle connessioni disponibili) quando la query è stata completata;
- quando una connessione cade (per esempio perché ha superato il tempo massimo di inattività) essa viene rimossa dalla coda delle connessioni disponibili.

Utilizzando tale sistema è possibile ridurre notevolmente il tempo di risposta quando numerose richieste sopraggiungono contemporaneamente; ciò è dovuto al fatto che per servire i Client ed effettuare le query, si usufruisce delle connessioni disponibili senza doverne stabilire di nuove. Tale sistema permette di avere un numero di connessioni al database aperte e disponibili per esser utilizzate che varia al variare del flusso di richieste:

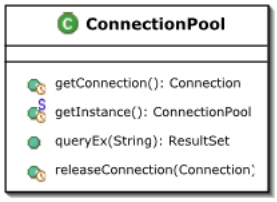


Figura 2.16: Diagramma della classe *ConnectionPool*

- all'aumentare del numero di richieste che sopraggiungono, nuove connessioni vengono istanziate e mantenute aperte e condivise per le successive richieste;
- quando il numero delle richieste diminuisce per un tempo significativo, le connessioni vengono poi chiuse e tolte dalla coda;

Al momento non è ancora stata creata una procedura che si occupi di chiudere le connessioni dopo che sia passato un determinato lasso di tempo e, la chiusura delle stesse, viene lasciata gestire al database in base al superamento del tempo massimo di inattività da esso stabilito. Per implementare tale tecnica sono state create due classi: ConnectionPool che si occupa di realizzarne il meccanismo e ConnectionPoolException per la gestione delle eventuali eccezioni che potrebbero verificarsi.

Classe *ConnectionPool* Nella classe *ConnectionPool* è realizzato il meccanismo della Pool di connessioni e tramite essa un *Profile Manager* può richiedere di effettuare una query al database. Essa implementa il design pattern Singleton per garantire che una sola istanza della classe venga creata ed utilizzata ed ha quindi un costruttore privato che non può essere invocato, implementando, per l'ottenimento dell'istanza della classe, il metodo statico getInstance().
Questa classe ha quindi i seguenti metodi:

- `public static synchronized ConnectionPool getInstance() throws ConnectionPoolException`
  metodo statico che, seguendo pattern Singleton, restituisce una sola

istanza di questa classe. In fase di inizializzazione della classe viene tentato il caricamento del driver per l'accesso al database e viene sollevata una *ConnectionPoolException* nel caso in cui ciò non sia possibile. È di tipo sincronizzato per non permettere che più istanze della classe vengano create contemporaneamente;

- `public synchronized Connection getConnection()`
  `throws ConnectionPoolException`
  restituisce una connessione libera prelevandola dalla coda oppure, se non ci sono connessioni disponibili, creandone una nuova con una chiamata al metodo privato newConnection(). È di tipo sincronizzato per non permettere che una stessa connessione venga assegnata a più Client e solleva una ConnectionPoolException nel caso in cui non sia possibile ottenere una connessione.;

- `public synchronized void releaseConnection(Connection con)`
  metodo sincronizzato che rilascia una connessione inserendola nella coda delle connessioni libere;

- `public ResultSet queryEx(String queryStr)`
  `throws ConnectionPoolException`
  metodo eseguire una query: si occupa di trovare una connessione libera su cui poterla effettuare e solleva una ConnectionPoolException nel caso non sia possibile ottenerla o nel caso in cui la query sia errata, questo metodo non è sincronizzato in quanto per svolgere il proprio compito utilizza i metodi sopracitati che sono già sincronizzati;

2.3 Reingegnerizzazione del modulo di aggregazione dati

Il lavoro eseguito sul *Context Provider* nell'ambito di questa tesi ha riguardato il sottomodulo di aggregazione dati *Merge* che è stato completamente riscritto ed il sottomodulo *net* per la comunicazione con i *Profile Manager*. In questa sezione ci occuperemo del sottomodulo me*Merge*tralasciando come questo comunichi con i profile manager ed incentrando la discussione sull'implementazione degli algoritmi di aggregazione dei profili e delle policies. La sequenza con cui si è giunti alla soluzione attuale è la seguente:

1. implementazione delle classi entità per la rappresentazione delle direttive per la risoluzione dei conflitti;

2. ideazione ed implementazione, con criteri di efficienza, di un algoritmo di aggregazione dei profili che sfrutti la libreria profiling (creata sempre nell'ambito di questa tesi) per eseguire l'aggregazione di questi sulla base delle direttive per la risoluzione dei conflitti;

3. ideazione ed implementazione, con criteri di efficienza, di un algoritmo che sulla base delle direttive esegua la pesatura delle policies e serializzi programma logico compatibile con l'attuale versione dell'*Inference Engine*.

2.3.1 Modellazione delle direttive

Una direttiva è, come anticipato nel Capitolo 2, un formalismo per assegnare delle priorità agli attributi provenienti dai vari *Profile Manager*, al fine di risolvere possibili conflitti che vengano ad evidenziarsi. Essa è identificata dalla parola-chiave "setPriority" e si compone di una serie di parametri tra cui: Componente/i e Attributo/i interessati dalla direttiva; una lista di *Profile Manager* in ordine di priorità. Queste direttive guidano il modulo di aggregazione dei profili nello svolgimento del suo compito. La priorità di una

direttiva in un elenco di n direttive, è proporzionale alla posizione, così che la direttiva in posizione n ha priorità n, maggiore della direttiva in posizione *n-1* e così via. La prima direttiva di aggregazione è quindi considerata avere la priorità minima, mentre l'ultima direttiva di aggregazione è considerata avere priorità massima.

Il linguaggio per la rappresentazione delle direttive di aggregazione è stato esteso per tenere conto anche del vocabolario di ogni componente e attributo. La sintassi per l'espressione di una direttiva si è quindi trasformata da:

$$\text{setPriority NomeComponente}_h \text{ / NomeAttributo}_j = (\text{ProfileManager}_1,\text{ProfileManager}_2,\text{ProfileManager}_n)$$

a:

$$\text{setPriority Vocabolario}_i\#\text{NomeComponente}_h \text{ / Vocabolario}_k\#\text{NomeAttributo}_j = (\text{ProfileManager}_1,\text{ProfileManager}_2,\text{ProfileManager}_n)$$

Atta ad esprimere che per tutti i valori di profilo il cui componente è definito nel Vocabolario_i ed ha il nome NomeComponente_h ed il cui attributo è definito nel vocabolario Vocabolario_k ed ha il nome NomeAttributo_j. La priorità per l'aggregazione dei profili è:

1. ProfileManager_1
2. ProfileManager_2
3. ProfileManager_n

Ciò, in primo luogo consente di distinguere Componenti e Attributi omonimi definiti in vocabolari diversi. In secondo luogo (dato che al posto di ogni Componente o Attributo o Vocabolario continua ad essere accettato la wild-card "asterisco" ("*") avente il significato di "tutti") diventa inoltre possibile definire delle direttive come:

$$\text{setPriority Vocabolario}_i\#^* \text{ / Vocabolario}_k\#^* = (\text{ProfileManager}_1,\text{ProfileManager}_2,\text{ProfileManager}_n)$$

Atta ad esprimere che per tutti i valori di profilo il cui componente, indipendentemente dal nome, è definito nel Vocabolario$_i$ ed il cui attributo, indipendentemente dal nome, è definito nel vocabolario Vocabolario$_k$ la priorità per l'aggregazione dei profili è: ProfileManager$_1$, ProfileManager$_2$, ProfileManager$_n$.

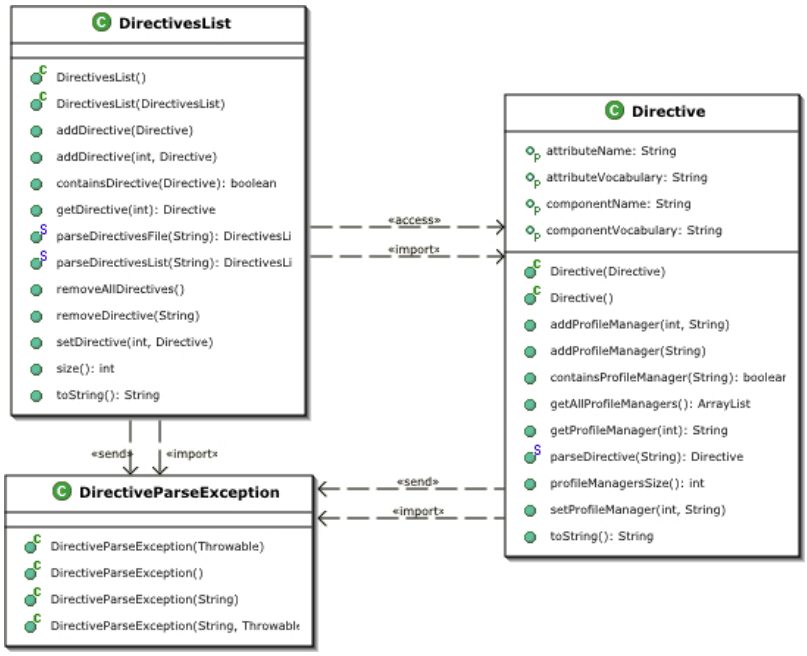


Figura 2.17: Diagramma del package *directives*

Per modellare le direttive è stato creato il sottopackage *directives* contenente tre classi:

- *Directive*, atta a rappresentare e gestire una singola direttiva;
- *DirectivesList*, atta a rappresentare e gestire un elenco di direttive;

- *DirectiveParseException* avente lo scopo di modellare le possibili eccezioni in fase di parsing delle direttive. Questa classe estende la *java.lang.Exception;*

Le prime due classi sono tra loro in relazione di composizione, ovvero un oggetto della classe *DirectivesList* si compone di uno o più oggetti della classe *Directive*, mentre *DirectiveParseException* viene sollevata dalle precedenti classi in tutti i casi in cui può avvenire un errore di parsing delle direttive.

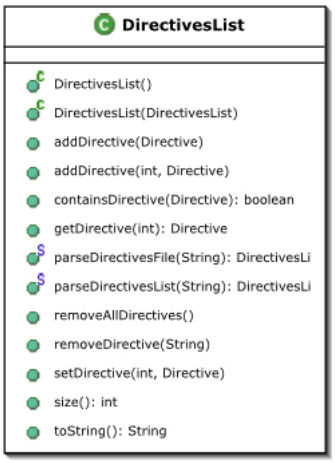


Figura 2.18: Diagramma della classe *DirectivesList*

Classe *DirectivesList* Questa classe ha il compito di gestire un elenco di direttive. Per la gestione di tale elenco è stata utilizzata anche in questo caso la classe *java.util.ArrayList*, in quanto era necessaria una struttura dati dinamicamente ridimensionabile a tempo di esecuzione ed indicizzata semplicemente tramite un intero. Questa classe ha due costruttori: uno che inizializza e restituisce una elenco di direttive inizializzato e vuoto ed uno che restituisce una copia dell'oggetto della classe *DirectivesList* passatagli come parametro. Essa implementa inoltre i seguenti metodi utili per la gestione dell'elenco ed il processing delle direttive:

- `public void addDirective(Directive directive)`
  aggiunge una direttiva infondo alla lista delle direttive;

- `public void addDirective(int index, Directive directive)`
  aggiunge una direttiva, nella posizione specificata dall'indice. La direttiva correntemente in quella posizione verrà scalata a destra di una posizione;

- `public boolean containsDirective(Directive directive)`
  restituisce *true* se la direttiva che riceve come parametro è contenuta nell'elenco;

- `public Directive getDirective(int index)`
  restituisce la direttiva memorizzata nella posizione specificata dall'indice;

- `public static DirectivesList parseDirectivesFile(String nomeFile) throws DirectiveParseException`
  per processare un file di testo contenente direttive, ogni direttiva deve essere su di una riga diversa. Solleva un'eccezione nel caso in cui una direttiva non rispetti la sintassi del linguaggio, o nel caso in cui non riesca ad aprire il file memorizzato su file-system;

- `public static DirectivesList parseDirectivesList(String directivesList) throws DirectiveParseException`
  per processare un file di testo contenente direttive, ogni direttiva deve essere su di una riga diversa. Solleva un'eccezione nel caso in cui una direttiva non rispetti la sintassi del linguaggio;

- `public void removeAllDirectives()`
  rimuove tutte le direttive dall'elenco;

- `public void removeDirective(String index)`
  rimuove la direttiva specificata dall'indice;

- `public void setDirective(int index, Directive directive)`
  sostituisce la direttiva nella posizione specificata con la direttiva specificata;

- `public int size()`
  restituisce la dimensione della lista di direttive;

- `public String toString()`
  serializza una stringa atta a rappresentare le direttive. L'output di questo metodo potrebbe a sua volta esser dato in input al metodo parseDirectivesList.

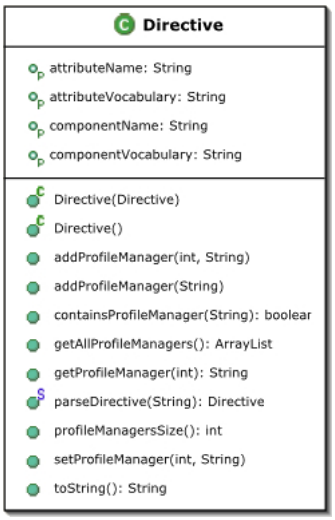


Figura 2.19: Diagramma della classe *Directive*

Classe *Directive* Questa classe ha lo scopo di rappresentare una direttiva ed è sostanzialmente una classe entità composta da diverse proprietà leggibili e scrivibili attraverso gli appropriati metodi get e set. Essa ha due costruttori: uno che inizializza e restituisce una direttiva vuota ed uno che restituisce una copia della direttiva passatagli come parametro formale. Le proprietà leggibili e scrivibili tramite gli appropriati metodi get e set sono appunto quelle relative a componente e attributo interessati dalla regola, ovvero:

- `attributeName:String`
  rappresenta il nome dell'attributo della direttiva;

- `attributeVocabulary:String`
  rappresenta il vocabolario dell'attributo della direttiva;

- `componentName:String`
  rappresenta il nome del componente della direttiva;

- `componentVocabulary:String`
  rappresenta il vocabolario del componente della direttiva;

Oltre ai metodi per settare le suddette proprietà questa classe implementa i seguenti metodi:

- `public void addProfileManager(String profileManager)`
  aggiunge un *Profile Manager* infondo alla lista di priorità dei *Profile Manager*;

- `public void addProfileManager(int index, String profileManager)`
  aggiunge un *Profile Manager*, nella posizione specificata dall'indice. Il *Profile Manager* correntemente in quella posizione verrà scalato a destra di una posizione;

- `public boolean containsProfileManager(String profileManager)`
  restituisce *true* se nella lista è presente un determinato *Profile Manager*;

- `public ArrayList getAllProfileManagers()`
  restituisce un'ArrayList contenente tutti i *Profile Manager* della direttiva;

- `public String getProfileManager(int index)`
  restituisce il *Profile Manager* memorizzato in una determinata posizione;

- `public static Directive parseDirective(String directiveString) throws DirectiveParseException`
  processa una direttiva scritta in forma testuale restituendo un oggetto della classe *Directive*. Solleva un'eccezione nel caso in cui la direttiva non rispetti la sintassi del linguaggio;

- `public int profileManagersSize()`
  restituisce il numero di *Profile Manager* presenti nella direttiva corrente;

- `public void setProfileManager(int index, String profileManager)`
  data una determinata posizione già occupata, sostituisce il *Profile Manager*, con quello passato come argomento;

- `public String toString()`
  serializza una stringa atta a rappresentare una direttiva. Tale stringa può a sua volta essere ri-processata dal metodo parseDirective.

2.3.2 Aggregazione dei profili e risoluzione dei conflitti

Come già anticipato nel Capitolo 2, esistono cinque principali categorie di conflitti che possono verificarsi nel momento in cui si cerca aggregare profili provenienti da diverse entità in un unico profilo aggregato:

1. *Conflitto tra valori espliciti per lo stesso attributo forniti da due differenti entità quando non c'è nessuna policy*;

2. *Conflitto tra uno specifico valore per un attributo e una policy, data dalla stessa entità, da cui potrebbe derivare un nuovo valore per l'attributo*;

3. *Conflitto tra un valore esplicito per un attributo e una policy, data da un entità differente, da cui potrebbe derivare un nuovo valore per l'attributo*;

4. *Conflitto tra due policies date da due differenti entità per uno specifico valore di un attributo*;

5. *Conflitto tra due regole date dalla stessa entità per uno specifico valore di un attributo.*

Tutti questi conflitti vengono risolti dalla classe *LogicMerge2* mediante due algoritmi: uno che effettua l'aggregazione dei profili CC/PP ed un altro algoritmo per la pesatura delle policies.

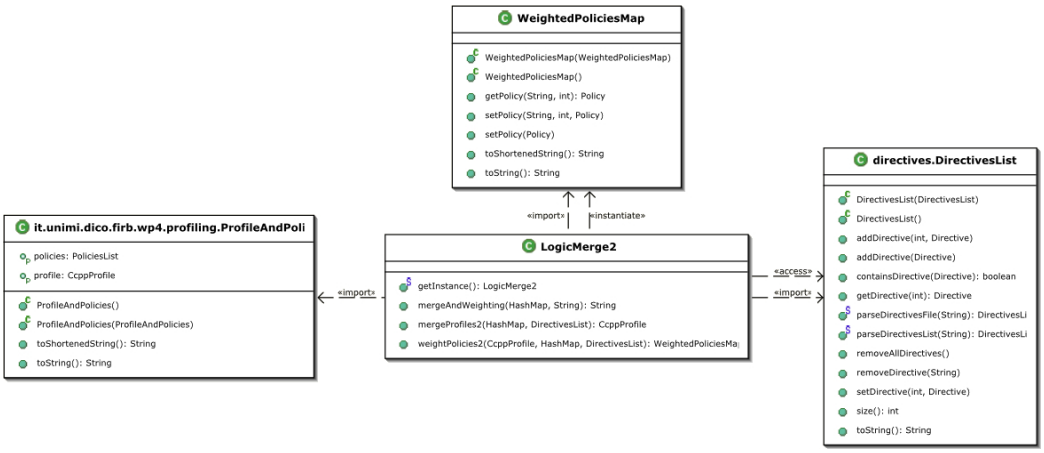


Figura 2.20: Diagramma del package *merge*

Le classi necessarie per l'esecuzione dell'algoritmo di aggregazione dati sono state raggruppate in un package chiamato *merge* contenente la sopracitata classe *LogicMerge2* ed una classe chiamata *WeightedPoliciesMap*, avente lo scopo di indicizzare tramite una mappa tutte le policy risultanti dall'operazione di aggregazione dei profili e pesatura delle policy.

Classe *WeightedPoliciesMap* Questa classe si occupa di catalogare le policies generate dal modulo *Merge*. Al momento oltre all'algoritmo di pesatura delle policies (che può anche farne a meno, serializzando solamente una stringa contenente il programma logico) non vi è nessuna entità a farne uso; tuttavia essa è stata progettata per possibili utilizzi futuri. Le policies memorizzate in questa mappa vengono indicizzate mediante un doppio indice composto dalla risorsa della policy (vocabolario#nomeComponente| vocabolario#nomeAttributo) e dal peso associato alla policy. È così possibile ricercare una specifica policy in base al peso ed in base al componente|attributo contenuto nel predicato dell'unico atomo della testa. Successivamente è pos-

Figura 2.21: Diagramma della classe *WeightedPoliciesMap*

sibile manipolare la policy così ottenuta, tramite i metodi della classe *Policy*, per esempio per ottenere il *Profile Manager* di provenienza.

Questa classe ha due costruttori: uno che restituisce un oggetto della classe *WeightedPoliciesMap* inizializzato e vuoto ed uno che restituisce una copia dell'oggetto della classe *WeightedPoliciesMap* che riceve come parametro formale. Essa implementa inoltre i seguenti metodi:

- `public Policy getPolicy(String head, int weight)`
  restituisce la policy memorizzata con gli indici: head, in cui è specificato il componente e l'attributo cui si riferisce (mediante una stringa composta da vocabolario#nomeComponente|vocabolario#nomeAttributo) ed il peso della policy che si sta cercando;

- `public void setPolicy(Policy policy)`
  aggiunge una policy alla mappa delle policies, risorsa e peso vengono automaticamente recuperate dalla policy;

- `public void setPolicy(String headResource, int weight, Policy policy)`
  aggiunge una policy alla mappa delle policies, specificandone risorsa e peso;

- `public String toShortenedString()`
  serializza una stringa, abbreviata dei vocabolari di componente/attributo, atta a rappresentare il programma logico costituito dalle policies;

- `public String toString()`
  serializza una stringa atta a rappresentare il programma logico costituito dalle policy.

Classe *LogicMerge2*

La classe *LogicMerge2* ha il compito di eseguire l'operazione di aggregazione dei profili CC/PP e sostituisce la classe *LogicMerge* che effettuava la medesima operazione sfruttando la libreria *jena* per la manipolazione dell'XML. La classe attuale svolge invece il proprio compito sfruttando la libreria *profiling*, precedentemente creata "ad-hoc", attraverso due algoritmi: l'algoritmo di aggregazione dei profili CC/PP che, guidato dalle direttive di aggregazione è responsabile della soluzione della prima delle cinque categorie di conflitti; l'algoritmo di pesatura delle policies che è responsabile della soluzione delle altre quattro categorie di conflitti. Questi algoritmi costituiscono la parte centrale del lavoro di questa tesi, nonché una delle caratteristiche centrali dell'architettura proposta. La classe *LogicMerge2* implementa il design pattern singelton in modo che esista una e una sola istanza condivisa della stessa.

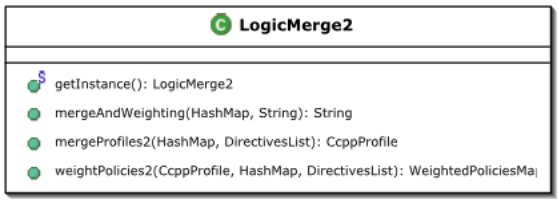


Figura 2.22: Diagramma della classe *LogicMerge2*

Essa, di conseguenza non ha costruttori pubblici, ma è possibile ottenere l'istanza della classe tramite il metodo statico getInstance(). Oltre al metodo precedentemente citato, questa classe implementa i seguenti metodi:

- `public CcppProfile mergeProfiles2(HashMap profiles, DirectivesList resolutionDirectives)`
  contiene l'algoritmo per l'aggregazione dei profili (che verrà più avanti

spiegato), esso ha come parametri: una mappa contenente i profili (di tipo *ProfileAndPolicies*) da aggregare e una lista di direttive;

- `public WeightedPoliciesMap weightPolicies2 (CcppProfile mergedProfile, HashMap profiles, DirectivesList resolutionDirectives)`
  contiene l'algoritmo di pesatura delle policies (che verrà spiegato più avanti), esso ha come parametri: il profilo aggregato precedentemente creato, una mappa di profili (di tipo *ProfileAndPolicies*) e una lista di direttive;

- `public String mergeAndWeighting(HashMap profiles, String directivesFilePath)`
  invoca in sequenza i metodi mergeProfiles2() e weightPolicies2() per ottenere un programma logico da passare in input all'*Inference Engine*. La stringa restituita da questo metodo è ottenuta invocando il metodo toShortenedString classe *WeightedPoliciesMap* che restituisce un programma logico abbreviato dei vocabolari.

Algoritmo di aggregazione dei profili CC/PP

L'algoritmo di aggregazione dei profili CC/PP, nell'ambito di questa tesi, è stato riprogettato e reimplementato al fine di sfruttare la libreria per la rappresentazione e gestione dei profili CC/P, la cui progettazione ed implementazione è esposta nelle precedenti sezioni.

Esso si occupa della risoluzione del primo tipo di conflitto, tra valori espliciti per lo stesso attributo forniti da due differenti entità. L'aggregazione dei profili è guidata dalle direttive di aggregazione per cui, per ogni direttiva, viene verificato se nei profili provenienti dai *Profile Manager* specificati dalla direttiva sono presenti dei Componenti/Attributi il cui nome e vocabolario corrispondono a quelli specificati dalla direttiva. In caso negativo, si procede all'analisi della direttiva successiva. In caso affermativo l'azione da intraprendere dipende dal tipo dell'attributo:

- se l'attributo è di tipo semplice ("rdf:Sin") il valore da assegnare a tale attributo nel profilo aggregato è quello proveniente dal primo *Profile Manager* della lista di *Profile Manager* specificata dalla direttiva;

- se l'attributo è di tipo "rdf:Bag" i valori da assegnare a tale attributo nel profilo aggregato sono tutti quelli provenienti dai *Profile Manager* della lista di *Profile Manager*. Se ci sono dei valori duplicati solo la prima occorrenza del valore è presa in considerazione;

- se l'attributo è di tipo "rdf:Seq" i valori da assegnare a tale attributo nel profilo aggregato sono tutti quelli provenienti dai *Profile Manager* della lista di *Profile Manager*, ordinati secondo l'ordine di occorrenza dei *Profile Manager* nella lista di *Profile Manager* della direttiva. Anche in questo caso se ci sono dei valori duplicati solo la prima occorrenza (sempre secondo l'ordine di occorrenza dei ProfileManager nella lista di *Profile Manager* della direttiva) del valore è presa in considerazione.

L'assenza in una direttiva di uno dei *Profile Manager* che si sta considerando per l'aggregazione dei profili, significa che per il/i Componente/i e Attributo/i specificato/i dalla direttiva, i valori provenienti dal quel determinato *Profile Manager*, non devono mai essere utilizzati. Tale caratteristica è implementata marcando come valutati (senza che siano inseriti nel profilo aggregato) tutti gli attributi provenienti da *Profile Manager* non citati dalla direttiva, ma che è necessario considerare per l'aggregazione dei profili.

Di seguito è riportato l'algoritmo per l'aggregazione dei profili CC/PP. Esso è diviso in tre parti, l'algoritmo principale chiamato *mergeProfiles2*, e due sottoprocedure chiamate *searchAttribute* e *storeAttribute*.

Algorithm 2 mergeProfiles2

Require: Una mappa contenente i profili da aggregare $ProfilesMap$ ed una lista di direttive $DirectivesList$.

Ensure: Un profilo aggregato secondo le direttive di aggregazione.

```
{L'iterazione verrà fatta dall'ultima alla prima}
for all Directive_x ∈ DirectivesList do
  for all ProfileManager_y ∈ (Directive_x.getPMs() ∪
  ProfilesMap.getPMs()) do
    CcppProfile_y ⇐ ProfilesMap.getProfile(ProfileManager_y);
    if Directive_x.componentContainsWildeCard() then
      for all Component_z ∈ CcppProfile_y do
        if Component_z.matches(Directive_x) then
          searchAttribute(mergedCcppProfile, Component_z,
          Directive_x, ProfileManager_y);
        end if
      end for
    else
      Component ⇐ CcppProfile_y.getComponent(Directive_x);
      if Component! = null then
        searchAttribute(mergedCcppProfile, Component, Directive_x,
        ProfileManager_y);
      end if
    end if
  end for
end for
return mergedCcppProfile
```

Notazione algoritmo 2

Molti dei metodi citati (come $Component_z.matches(Directive_x)$ sono delle astrazioni per facilitare la comprensione dell'algoritmo.

- $ProfilesMap$ è una mappa contenente tutti i profili CC/PP (rappresentati come oggetti della classe *CcppProfile*), indicizzati con il nome del *Profile Manager* da cui provengono;
- $Directive_x$ è un oggetto della classe *Directive* atto a rappresentare la direttiva attualmente considerata;
- *DirectivesList* è un oggetto della classe *DirectiveList* contenente un elenco ordinato di direttive di aggregazione $Directive_x$. L'iterazione verrà fatta partendo dall'ultima direttiva (con priorità maggiore);
- $ProfileManager_y$ è una stringa di caratteri contenente il nome del *Profile Manager* da prendere in considerazione.
- $Directive_x.getPMs()$ ritorna un array di stringhe di caratteri contenenti i nomi dei *Profile Manager* specificati dalla direttiva $Directive_x$, in ordine di occorrenza;
- $ProfilesMap.getPMs()$ ritorna un array di stringhe di caratteri contenenti i nomi dei *Profile Manager*, che vengono usati come chiavi nella mappa di profili da aggregare $ProfilesMap$;
- $Directive_x.componentContainsWildeCard()$ restituisce *true* se il componente definito dalla direttiva $Directive_x$ contiene almeno una Wildecard "asterisco" ed è quindi necessaria un'iterazione per recuperare tutti i component che corrispondono alla direttiva;
- $ProfilesMap.getProfile(ProfileManager_y)$ restituisce un oggetto della classe *CcppProfile* atto a rappresentare il profilo CC/PP proveniente dal *Profile Manager* $ProfileManager_y$;

- $CcppProfile_y$ è il profilo CC/PP rappresentato tramite un oggetto della classe *CcppProfile*;

- $Component_z$ è un oggetto della classe *Component*, atto a rappresentare un componente del profilo CC/PP attuale $CcppProfile_y$;

- $Component_z.matches(Directive_x)$ restituisce `true` se e solo se il componente attuale $Component_z$ ha lo stesso vocabolario della direttiva $Directive_x$ (o il vocabolario nella direttiva è settato ad *) e ha lo stesso nome della direttiva (o il nome nella direttiva è settato ad *);

- $searchAttribute(mergedCcppProfile, Component_z, Directive_x, ProfileManager_y)$ invoca il metodo searchAttribute, incaricato di cercare tutti gli attributi nell'attuale Componente $Component_z$ che soddisfano la direttiva $Directive_x$ e memorizzarli nel profilo aggregato $mergedCcppProfile$, salvando anche il profile manager da cui provengono $ProfileManager_y$;

- $CcppProfile_y.getComponent(Directive_x)$ invoca il metodo getComponent sull'oggetto della classe *CcppProfile* $CcppProfile_y$, che restituisce un oggetto della classe *Component*, appartenete al profilo, il cui vocabolario e nome coincidono con il vocabolario ed il nome del componente specificato dalla direttiva $Directive_x$. Se il componente non viene trovato, il metodo restituisce il valore `null`;

- $mergedCcppProfile$ è un oggetto della Classe *CcppProfile*, atto a rappresentare il profilo CC/PP aggregato. Inizialmente è vuoto.

Algorithm 3 searchAttribute

Require: Un profilo $mergedCcppProfile$ in cui inserire i valori degli attributi, un componente $Component$ in cui cercare gli attributi che soddisfano la direttiva, una direttiva $Directive_x$ ed il *Profile Manager* di provenienza del componente $ProfileManager_y$.

Ensure: Aggiunge al profilo aggregato tutti gli attributi di un componente che matchano la direttiva.

if $Directive_x.attributeContainsWildCard()$ **then**
  for all $Attribute_i \in Component_z$ **do**
    if $(Attribute_i.matches(Directive_x)$&&$Attribute_i.hasNotBeenEvalued)$ || $(Attribute_i.isNotSimple$ &&$Attribute_i.hasNotBeenEvalued(ProfileManager_y))$ **then**
      if $Directive_x.containsProfileManager(ProfileManager_y)$ **then**
        $storeAttribute(mergedCcppProfile, Component_z, Attribute_i, ProfileManager_y)$;
      end if
      $Attribute_i.markAsEvalued$;
      $Attribute_i.markAsEvalued(ProfileManager_y)$;
    end if
  end for
else
  $Attribute \Leftarrow Component.getAttribute(Directive_x)$;
  if $(Attribute != null$&&$Attribute.hasNotBeenEvalued)$ || $(Attribute.isNotSimple$ &&$Attribute.hasNotBeenEvalued(ProfileManager_y))$ **then**
    if $Directive_x.containsProfileManager(ProfileManager_y)$ **then**
      $storeAttribute(mergedCcppProfile, Component, Attribute, ProfileManager_y)$
    end if
    $Attribute.markAsEvalued$;
    $Attribute.markAsEvalued(ProfileManager_y)$;
  end if
end if

Teorema 1 *La complessità di questo algoritmo è lineare rispetto al numero di valori di profilo, poiché ogni attributo (e quindi ogni valore) viene considerato una sola volta.*

Notazione algoritmo 3

Molti dei metodi citati (come $Attribute_i.matches(Directive_x)$) sono delle astrazioni per facilitare la comprensione dell'algoritmo.

- $Directive_x.attributeContainsWildCard()$ restituisce `true` se l'attributo definito dalla direttiva $Directive_x$ contiene almeno una Wildecard "asterisco" ed è quindi necessaria un'iterazione per recuperare tutti i component che corrispondono alla direttiva;

- $Attribute_i$ è un oggetto della classe *Attribute*, atto a rappresentare un attributo appartenente al componente attuale $Component_z$;

- $Attribute_i.matches(Directive_x)$ restituisce `true` se e solo se l'attributo $Attribute_i$ ha lo stesso vocabolario della direttiva $Directive_x$(o il vocabolario nella direttiva è una Wildecard) e l'attributo ha lo stesso nome della direttiva $Directive_x$(o il nome nella direttiva è una Wildecard "asterisco");

- $Attribute_i.hasNotBeenEvalued$ restituisce `true` se e solo se l'attributo $Attribute_i$ non è ancora stato marcato come "valutato" durante l'algoritmo 2 mediante la keyword $Attribute_i.markAsEvalued$;

- $Attribute_i.isNotSimple$ restituisce `true` se e solo se l'attributo $Attribute_i$ non è di tipo semplice, ovvero è di tipo "rdf:Seq" o "rdf:Bag";

- $Attribute_i.hasNotBeenEvalued(ProfileManager_y)$ restituisce `true` se e solo se l'attributo $Attribute_i$ non è mai stato marcato come "valutato" per il *Profile Manager* specificato come argomento $ProfileManager_y$, indipendentemente dal fatto che sia già stato valutato su altri *Profile Manager*;;

- $Directive_x.containsProfileManager(ProfileManager_y)$ restituisce `true` se il *Profile Manager* attuale: $ProfileManager_y$ è nella lista specificata dalla direttiva $Directive_x$ dei profile manager da contattare (invoca il metodo containsProfileManager della classe *Directive*);

- $storeAttribute(mergedCcppProfile, Component, Attribute, ProfileManager_y)$ invocazione della sottoprocedura `storeAttribute`, passandogli come argomento, l'attuale profilo aggregato $mergedCcppProfile$, il componente $Component$ e l'attributo $Attribute$ che hanno verificato la direttiva ed il nome del *Profile Manager* in cui sono stati trovati $ProfileManager_y$;

- $Attribute_i.markAsEvalued$ marca l'attributo $Attribute_i$ come già valutato;

- $Attribute_i.markAsEvalued(ProfileManager_y)$ marca l'attributo $Attribute_i$ come già valutato per il *Profile Manager* corrente $ProfileManager_y$;

- $Component$ è una istanza della classe $Component$;

- $Component.getAttribute(Directive_x)$ invoca il metodo getAttribute sull'oggetto della classe *Component* $Component$, che restituisce un oggetto della classe *Attribute*, appartenente al componente $Component$, il cui vocabolario e nome coincidono con il vocabolario ed il nome del componente specificato dalla direttiva $Directive_x$. Se nessun attributo viene trovato il metodo restituisce il valore `null`.

Algorithm 4 storeAttribute

Require: Un profilo aggregato $mergedCcppProfile$ a cui aggiungere il/i valore/i dell'attributo *Attribute*, l'attributo *Attribute* da memorizzare, il componente *Component* contenente l'attributo *Attribute*, ed il *Profile Manager* $ProfileManager_y$) da cui proviene l'attributo da memorizzare.

Ensure: Se necessario aggiunge al profilo aggregato $mergedCcppProfile$ un nuovo componente ed un nuovo attributo aventi vocabolario e nome uguali al vocabolario e nome del componente *Component* e attributo *Attribute* passati come argomento. Aggiunge inoltre al nuovo attribuo $mergedCcppProfile$ il/i valore/i dell'attributo *Attribute*, se necessario ordinandoli ed eliminando i duplicati.

$finalAttribute \Leftarrow null$
$finalComponent \Leftarrow mergedCcppProfile.get(Component)$;
if $finalComponent == null$ **then**
    $finalComponent \Leftarrow newComponent(Component)$;
    $mergedCcppProfile.addComponent(finalComponent)$;
else
    $finalAttribute \Leftarrow finalComponent.getAttribute(Attribute)$;
end if
if $finalAttribute! = null$&&$(Attribute.isNotSimple)$ **then**
    $finalValuesSize \Leftarrow finalAttribute.valuesSize()$;
    $jump \Leftarrow 0$;
    for all $Value_k \in Attribute$ **do**
        if $!finalAttribute.containsValue(Value_k)$ **then**
            $Value_k.setProfileManager(ProfileManager_y)$;
            $storePosition \Leftarrow finalValuesSize + k - jump$;
            $finalAttribute.addValue(storePosition, Value_k)$;
        else
            $jump \Leftarrow jump + 1$;
        end if
    end for
else
    $finalAttribute \Leftarrow Attribute$;
    $finalAttribute.setProfileManager(ProfileManager_y)$;
end if
$finalComponent.setAttribute(finalAttribute)$

Notazione algoritmo 4

Molti dei metodi citati (come $Attribute.isNotSimple$) sono delle astrazioni per facilitare la comprensione dell'algoritmo.

- $mergedCcppProfile$ è il profilo aggregato fin'ora costruito dall'algoritmo 2;
- *Attribute* è una istanza della classe *Attribute* e rappresenta l'attributo i che si vorrebbe memorizzare nel profilo aggregato $mergedCcppProfile$;
- *Component* è una istanza della classe *Component* e rappresenta il componente in cui è contenuto l'attributo *Attribute* i che si vorrebbe memorizzare nel profilo aggregato $mergedCcppProfile$;
- $finalAttribute$ è una istanza della classe *Attribute* e rappresenta l'attributo che verrà memorizzato nel profilo aggregato $mergedCcppProfile$;
- $finalComponent$ è una istanza della classe *Component* che rappresenta il componente del profilo aggregato $mergedCcppProfile$ in cui verrà inserito il nuovo attributo $finalAttribute$;
- $mergedCcppProfile.get(Component)$ invoca il metodo getComponent sull'oggetto della classe *CcppProfile* $mergedCcppProfile$, che restituisce un oggetto della classe *Component*, appartenete al profilo, il cui vocabolario e nome coincidono con il vocabolario ed il nome del componente specificato del componente *Component*. Se il componente non viene trovato, il metodo restituisce il valore `null`;
- $newComponent(Component)$ crea un nuovo oggetto della classe *Component* i cui vocabolario e nome sono copiati da quelli del componente *Component*;
- $mergedCcppProfile.addComponent(finalComponent)$ aggiunge il componente $finalComponent$ al profilo aggregato $mergedCcppProfile$;

- $finalComponent.getAttribute(Attribute)$ invoca il metodo getAttribute sull'oggetto della classe *Component* $finalComponent$, che restituisce un oggetto della classe *Attribute*, appartenente al componente $finalComponent$, il cui vocabolario e nome coincidono con il vocabolario ed il nome del componente specificato dalla direttiva $Directive_x$. Se nessun attributo viene trovato il metodo restituisce il valore `null`;

- *Attribute.isNotSimple* restituisce `true` se l'attributo non è di tipo semplice, ovvero è un attributo del tipo rdf:Seq o rdf:Bag;

- $finalValuesSize$ è la dimensione dell'array di valori appartenenti all'attributo $finalAttribute$;

- $finalAttribute.valuesSize()$ restituisce la dimensione dell'array di valori appartenenti all'attributo $finalAttribute$;

- $jump$ è un contatore che memorizza il numero di valori identici già memorizzati nell'attributo $finalAttribute$;

- $Value_k$ è un oggetto della classe *Value*, atto a rappresentare un valore dell'attributo attuale $finalAttribute$;

- $Value_k.setProfileManager(ProfileManager_y)$ invoca il metodo setProfileManager dell'oggetto della classe value *Value* $Value_k$ al fine di memorizzare il *Profile Manager* $ProfileManager_y$ da cui proviene;

- $storePosition$ è la posizione in cui verrà memorizzato un valore;

- $finalAttribute.addValue(storePosition, Value_k)$ aggiunge all'attributo $finalAttribute$, nella posizione $storePosition$, il valore $Value_k$;

- $finalAttribute.setProfileManager(ProfileManager_y)$ invoca il metodo setProfileManager dell'oggetto della classe *Attribute* $finalAttribute$ al fine di memorizzare il ProfileManager $ProfileManager_y$ da cui proviene;

- $finalComponent.setAttribute(finalAttribute)$ invoca il metodo setAttribute sull'oggetto della classe *Component* $finalComponent$ che aggiunge a questo l'oggetto della classe *Attribute* $finalAttribute$.

Questo algoritmo ha solo lo scopo di esplicitare come viene effettuata l'unione dei nomi dei *Profile Manager* specificati da ogni singola direttiva, con i nomi di tutti i *Profile Manager* che forniscono un profilo.

Algorithm 5 $Directive_x.getPMs() \cup ProfilesMap.getPMs()$

Require: Due liste di *Profile Manager* $Directive_x.getPMs()$ e $ProfilesMap.getPMs()$, di cui la prima ordinata.

Ensure: Aggiunge alla lista dei *Profile Manager* da considerare, i *Profile Manager* non presenti tra quelli provenienti dalla direttiva.

Notazione: $ProfileManagersToConsider$ contiene i Profile Manager risultanti dall'operazione di unione: $Directive_x.getPMs() \cup ProfilesMap.getPMs()$.

$ProfileManagersToConsider \Leftarrow Directive_x.getPMs()$
for all $ProfileManager_w \in ProfilesMap$ **do**
  if $!ProfileManagersToConsider.contains(ProfileManager_w)$ **then**
    $ProfileManagersToConsider.add(ProfileManager_w)$;
  end if
end for
return $ProfileManagersToConsider$

Breve spiegazione dell'algoritmo L'algoritmo di aggregazione dei profili è guidato dalle direttive di aggregazione, esso esegue la seguente iterazione:

- partendo dall'ultima direttiva (con priorità maggiore) verso la prima, per ognuna di esse viene eseguita un'iterazione in ordine di priorità, su tutti i profili provenienti dai *Profile Manager* specificati dalla direttiva più i restanti *Profile Manager*.;

- per ognuno dei *Profile Manager* sopracitati, vengono recuperati i componenti e gli attributi che non siano stati marcati come valutati e che soddisfino la direttiva;

- per ogni attributo che non è stato ancora marcato come valutato (oppure che è stato marcato come valutato, ma non per quel *Profile Manager* ed è di tipo non semplice, ovvero rdf:Seq o rdf:Bag) e che soddisfa la direttiva, vengono intraprese le seguenti azioni:

  - se esso proviene da un *Profile Manager* compreso tra i profile manager specificati dalla direttiva:

    * se esso è di tipo non semplice (rdf:Seq o rdf:Bag) i suoi valori vengono memorizzati nell'attributo del profilo aggregato, eliminando i duplicati e assegnando ad essi un peso incrementale;

    * se esso è di tipo semplice (rdf:Sin) il suo valore è assegnato all'attributo del profilo aggregato;

  - esso viene marcato come valutato (per non essere mai più considerato). Marcare come valutati anche gli attributi provenienti da *Profile Manager* non citati dalla direttiva ha lo scopo di evitare che tali attributi (che secondo quanto sopra detto, non debbono mai essere considerati) vengano in seguito erroneamente memorizzati nel profilo aggregato.

La complessità di questo algoritmo è quindi lineare rispetto al numero di valori presenti nei profili da aggregare, poiché ogni attributo (e quindi ogni valore) viene considerato una sola volta.

Una delle caratteristiche di questo algoritmo è (come si può notare in riga [5] dell'algoritmo 2 ed in riga [1] dell'algoritmo 3) che esso si occupa di verificare se la direttiva contiene o meno la wildcard "asterisco", sia nel componente, che nell'attributo specificato. Ciò è fatto al fine di stabilire se sia necessario effettuare una ricerca esaustiva di tutti i componenti o attributi che la soddisfano. Nello specifico, l'algoritmo verifica se nella coppia vocabolario#nomeComponente specificata dalla direttiva sia presente una wildcard "asterisco". In caso affermativo esegue una iterazione completa su tutto l'insieme di componenti per recuperare quelli che la soddisfano. In caso contrario richiama il metodo della classe *CcppProfile*.getComponent(), per tentare di recuperare direttamente il componente e successivamente verifica che questo non sia nullo (se lo è significa che il profilo non conteneva il componente che si è tentato di recuperare). La stessa operazione viene fatta per recuperare tutti gli attributi del componente appena considerato che soddisfano la direttiva. Anche in questo caso l'algoritmo verifica se nella coppia vocabolario#nomeAttributo specificata dalla direttiva sia presente una wildcard "asterisco". In caso affermativo esegue una iterazione completa per recuperare tutti gli attributi che la soddisfano. In caso contrario richiama il metodo della classe *Component*.getAttribute() per tentare di recuperare direttamente l'attributo e successivamente verifica che questo non sia nullo (se lo è significa che il componente non conteneva l'attributo che si è tentato di recuperare). Questa distinzione viene fatta in quanto normalmente (e particolarmente nell'implementazione attuale) per cercare e recuperare un solo elemento all'interno di un insieme di elementi non viene eseguita una semplice iterazione su tutti gli elementi, ma vengono applicati algoritmi più efficienti.

Per la rappresentazione delle mappe di componenti e attributi del profilo è stata scelta (come già detto nella sezione relativa alla libreria *profil-*

ing) la classe *java.util.HashMap* che, secondo quanto dichiarato nelle API, garantisce un tempo costante per le operazioni di get() e put(). Nell'implementazione Java di questo algoritmo ci si avvantaggia molto di questa scelta perché ogni volta che una parte di una direttiva, relativa ad un componente o ad un attributo, non contiene nessuna wildcard "asterisco" è possibile tentare di recuperare direttamente il componente o l'attributo, appunto mediante l'operazione di get() che restituisce l'oggetto desiderato in un tempo costante. Successivamente viene eseguito un semplice controllo per verificare che l'oggetto recuperato non sia nullo. Solamente quando invece una parte di direttiva contiene una wildcard "asterisco", viene effettuata un iterazione completa che impiegherà un tempo direttamente proporzionale al numero di componenti o attributi del profilo che si sta analizzando.

Algoritmo di pesatura delle policies

L'algoritmo di pesatura delle policies ha il compito di risolvere il secondo, terzo, quarto e quinto tipo di conflitto, ovvero:

1. *Conflitto tra uno specifico valore per un attributo e una policy, data dalla stessa entità, da cui potrebbe derivare un nuovo valore per l'attributo*;

2. *Conflitto tra un valore esplicito per un attributo e una policy, data da un entità differente, da cui potrebbe derivare un nuovo valore per l'attributo*;

3. *Conflitto tra due policies date da due differenti entità per uno specifico valore di un attributo.*

4. *Conflitto tra due regole date dalla stessa entità per uno specifico valore di un attributo.*

Le azioni da intraprendere sono rispettivamente:

1. La policy ha priorità maggiore rispetto all'attributo.

2. Per la risoluzione del conflitto è necessario ricorrere alle direttive: se l'entità da cui arriva la policy ha priorità minore rispetto a quella da cui arriva il valore esplicito, allora la policy può essere ignorata, altrimenti essa viene considerata e se da essa scaturisce un nuovo valore, questo prevale su quello esplicito.

3. *S*imilmente al conflitto precedente, la priorità è stabilità dalle direttive, in base alla priorità delle entità da cui proviene la policy.

4. *L*a priorità assegnata alla direttiva mediante l'espressione $R_1 > R_2$ stabilisce quale delle due regole considerare, se nessuna priorità è stata impostata verrà utilizzato un ordinamento arbitrario.

La modellazione della priorità di una policy avviene mediante la proprietà *weight* cui è possibile accedere (attraverso gli opportuni metodi set e get()) per ogni policy. Il peso di una policy è definito come il peso associato al predicato nell'intestazione. Le policy con peso maggiore avranno priorità maggiore. Segue la descrizione Formale dell'algoritmo *weightPolicies*, a sua volta composto di una sottoprocedura chiamata *computePolicyWeight*.

Algorithm 6 weightPolicies

Require: Il profilo aggregato $mergedCcppProfile$; una mappa $policiesMap$ avente come chiave il nome di un *Profile Manager* e come valore una lista di policies $policiesList$ ordinata in base al peso; un elenco di direttive $directivesList$.

Ensure: Un elenco di policies pesate secondo le modalità descritte.

$weight \Leftarrow 0$;
$weightedPoliciesMap \Leftarrow newWeightedPoliciesMap$;
$mergePolicies \Leftarrow newMap$;
$policiesWeight \Leftarrow newMap$
for all $component_x \in mergedCcppProfile$ **do**
  for all $attribute_y \in component_x$ **do**
    $policy_z \Leftarrow newPolicy()$;
    for all $value_z \in attribute_y$ **do**
      $headParameter_z \Leftarrow headParameter_z + value_z$;
    end for
    $policyHead_z \Leftarrow newPolicyPart(component_x, attribute_y)$;
    $policy_z.setHead(policyHead_z)$;
    $policy_z.setWeight(weight)$;
    $mergePolicies.set(policy_z, attribute_y.getProfileManager())$;
    $weightedPoliciesMap.setPolicy(policy_z)$;
  end for
end for
for all $directive_i \in directivesList$ **do**
  {In questo caso l'iterazione verrà fatta dall'ultimo al primo}
  for $profileManager_j \in (directive_i.getPMs() \cup policiesMap.getPMs())$ **do**
    $policiesList_k \Leftarrow policiesMap.get(profileManager_j)$;
    for all $policy_h \in policiesList_k$ **do**
      if $policy_h.hasNotBeenEvalued()$ **then**
        if $policy_h.matches(Directive_i)$ **then**
          $profileManager_h \Leftarrow mergePolicies.get(policy_h)$
          if $profileManager_h != null$ **then**
            if $profileManager_j.isNotAfter(profileManager_h)$ **then**
              $weight_h \Leftarrow computePolicyWeight(policiesWeight, policy_h, profileManager_j)$;
              $policy_h.setWeight(weight_h)$;
              $weightedPoliciesMap.setPolicy(policy_h)$;
            end if
          else
            $weight_h \Leftarrow computePolicyWeight(policiesWeight, policy_h, profileManager_j)$;
            $policy_h.setWeight(weight_h)$;
            $weightedPoliciesMap.setPolicy(policy_h)$;
          end if
          $policy_h.markAsEvalued$;
        end if
      end if
    end for
  end for
end for

return $weightedPoliciesMap$;

Notazione algoritmo 6

Molti dei metodi citati (come $policy_h.matches(Directive_i)$) sono delle astrazioni per facilitare la comprensione dell'algoritmo.

- $weightedPoliciesMap$ è un oggetto della classe *WeightedPoliciesMap*;
- $policiesWeight$ è una mappa contenente: come chiave la risorsa (vocabolario#nomeComponente|vocabolario#nomeAttributo) di tutte le policies che sono state pesate; come valore, un'array di pesi associato a ciascuna di esse;
- $mergePolicies$ è una mappa contenente: come chiave la risorsa (vocabolario#nomeComponente|vocabolario#nomeAttributo) di tutte le policies che sono state derivate da un valore per un attributo del profilo aggregato. Come valore il *Profile Manager* di provenienza di ciascuna di esse;
- $policy_z$ è un oggetto della classe *Policy*;
- $headParameter_z$ il parametro della testa della policy $policy_z$;
- $policyHead_z$ è l'intestazione della policy $policy_z$;
- $newPolicyPart(component_x, attribute_y)$ costruisce una nuova PolicyPart avente un solo atomo di tipo CC/PP il cui vocabolario e nomeComponente è ottenuto dal componente $component_x$ ed il cui vocabolario e nomeAttributo è ottenuto dal'attributo $attribute_y$;
- $mergePolicies.set(policy_z, attribute_y.getProfileManager())$ aggiunge alla mappa delle policies derivate dal profilo aggregato una coppia chiave/valore in cui la chiave è il predicato dell'intestazione della policy $policy_z$ ed il cui valore è il *Profile Manager* di provenienza della policy, calcolato come $attribute_y.getProfileManager()$;
- $weightedPoliciesMap.setPolicy(policy_z)$ aggiunge alla mappa delle policies, la policies appena costruita;

- $policiesList_k$ è la lista di policies provenienti da un determinato *Profile Manager*;
- $policy_h.hasNotBeenEvalued()$ restituisce `true` se la policy non è ancora stata marcata come valutata;
- $policy_h.matches(Directive_i)$ restituisce `true` se la policy soddisfa la direttiva;
- $profileManager_h$ è il *Profile Manager* da cui proviene un eventuale policy derivata dal profilo aggregato, la cui risorsa (vocabolario#nomeComponente|vocabolario#nomeAttributo) coincide con quella della policy $policy_h$;
- $profileManager_j.isNotAfter(profileManager_h)$ restituisce true se il *Profile Manager* $profileManager_j$ da cui proviene la policy $policy_h$ non è successivo, all'interno dell'elenco di *Profile Manager* da contattare della direttiva, rispetto al *Profile Manager* $profileManager_h$ della policy derivata dal profilo aggregato;
- $weight_h$ è il peso da associare alla policy $policy_h$;
- $computePolicyWeight(policiesWeight, policy_h, profileManager_j)$ invocazione del metodo contenente l'algoritmo per il calcolo del peso della policy $policy_h$;
- $policy_h.markAsEvalued$; marca la policy $policy_h$ come valutata.

Algorithm 7 computePolicyWeight

Require: una mappa di pesi associati alle policies $policiesWeight$, la policy su cui effettuare il calcolo $currentPolicy$, ed il *Profile Manager* da cui proviene la policy $profileManager$

Ensure: Ritorna il peso da associare alla policy.

$weights[]$;
$originalWeight$;
$finalWeight$
$policyResource \Leftarrow policy.getResource()$;
$weights[] \Leftarrow policiesWeight.get(policyResource)$;
if $weights[] == null$ **then**
  $weights \Leftarrow newint[3]$;
  $weights[0] \Leftarrow 0$;
  $weights[1] \Leftarrow 1$;
  $weights[2] \Leftarrow profileManager$;
else
  if $profileManager! \Leftarrow weights[2]$ **then**
    $weights[1] \Leftarrow weights[0]$;
    $weights[2] \Leftarrow currentProfileManager$;
  end if
end if
$originalWeight \Leftarrow currentPolicy.getWeight()$;
if $originalWeight > weights[1]$ **then**
  $finalWeight \Leftarrow weights[0] + (originalWeight - weights[1])$;
  $weights[1] \Leftarrow originalWeight$;
else
  $finalWeight \Leftarrow weights[0] + 1$;
end if
if $finalWeight > weights[0]$ **then**
  $weights[0] \Leftarrow finalWeight$;
end if
$policiesWeight.put(policyResource, weights[])$;
return $finalWeight$

Notazione algoritmo 7

- *policyResource* è la risorsa della policy attuale (composta da vocabolario#nomeComponente|vocabolario#nomeAttributo), ovvero il predicato dell'unico atomo della testa della policy;
- $weights[]$ è un array di tre elementi, associato alla policy avente come risorsa la stessa risorsa della policy attuale *policyResource* ;
- $originalWeight$ è il peso originale della policy, questo viene stabilito dai *Profile Manager*;
- $finalWeight$ è il peso che verrà associato alla policy alla fine dell'operazione di pesatura;
- $weights[0]$ è il peso massimo, per una data policy raggiunto in un certo istante;
- $weights[1]$ è il peso massimo riscontrato tra i pesi originali $originalWeight$ delle policy;
- $weights[2]$ in questo elemento viene memorizzato il profile manager da cui proviene la policy;
- $policiesWeight.put(policyResource, weights[])$ aggiunge o sovrascrive l'array associato alla risorsa *policyResource*.

Breve spiegazione dell'algoritmo L'algoritmo di pesatura delle policies è anch'esso guidato dalle direttive di aggregazione ed esegue quindi le seguenti operazioni:

1. per ogni valore del profilo crea una policy senza corpo avente peso zero;
2. partendo dall'ultima direttiva (con priorità maggiore) verso la prima, per ognuna di esse viene eseguita un'iterazione su tutti i profili provenienti dai *Profile Manager* specificati dalla direttiva, più i restanti *Profile*

Manager. L'iterazione è eseguita in ordine inverso di priorità, ovvero partendo dall'ultimo *Profile Manager* fino ad arrivare al primo, ciò permette di assegnare alle policies recuperate, un peso via via maggiore;

3. per ognuno dei *Profile Manager* sopracitati, vengono recuperate tutte le policies che non siano ancora state marcate come valutate e che soddisfino la direttiva;

4. per ogni policy che non è stata ancora marcata come valutata e che soddisfa la direttiva:

   - se la policy proviene da un *Profile Manager* compreso tra i profile manager della direttiva, essa viene pesata e memorizzata nell'elenco finale di policies. Il peso assegnato alle policies è calcolato nel seguente modo:
     - alle policies provenienti da una stessa entità viene assegnato un peso pari a quello che era stato specificato dall'entità o, se due policies hanno peso uguale, arbitrariamente ad una di esse viene assegnato il peso dell'altra incrementato di uno;
     - alle policies provenienti da entità diverse viene assegnato un peso incrementale pari al peso massimo raggiunto per ogni policy nelle entità con priorità inferiore, più il peso calcolato secondo quanto specificato sopra;
   - essa viene marcata come valutata (per non essere mai più considerata). Marcare come valutate anche le policies provenienti da *Profile Manager* non citati dalla direttiva è necessario per evitare che tali policies vengano in seguito erroneamente memorizzate nell'elenco finale di policies.

La complessità di questo algoritmo è lineare rispetto al numero di policy da pesare, poiché ogni regola viene considerata una sola volta. Esso richiede che le policy provenienti da ogni *Profile Manager* siano già ordinate per peso. Questo perché è molto più efficiente eseguire l'ordinamento di tutte le policy

provenienti dai un *Profile Manager* una sola volta prima dell'esecuzione dell'algoritmo, piuttosto che eseguire tale ordinamento ad ogni iterazione, cioè per ogni *Profile Manager* di ogni direttiva. Nell'implementazione attuale, tale ordinamento viene fatto dal DBMS nel momento in cui viene effettuata la richiesta.

Purtroppo nella libreria J2SE di Java 1.4.2 l'unica mappa in grado di mantenere un ordinamento è la java.util.TreeMap che esegue però le operazioni di get() e put() in un tempo pari log(*numeroElementi*), perciò si è optato per un normale array indicizzato numericamente ed in particolare si è optato per la classe *java.util.ArrayList* che garantisce prestazioni più elevate. In questo caso però, non potendo sfruttare il metodo get() della classe *HashMap* per effettuare in modo efficiente la ricerca di una singola policy indicizzata tramite la coppia Componente|Attributo, non è neanche stata fatta nessuna differenziazione tra direttive contenenti la wildcard "asterisco" e direttive che riguardano una sola specifica coppia Componente|Attributo. La soluzione migliore a questo problema sarebbe implementare, per le direttive non contenenti la wildcard "asterisco", qualche tipo di ricerca della policy che sia più efficiente di una iterazione completa. In alternativa si potrebbe pensare di invertire l'approccio sopra presentato utilizzando, per gestire l'elenco di policy, la classe *java.util.TreeMap* al posto della classe *java.util.ArrayList* e confrontare i risultati ottenuti. Oltre a ciò è in fase di studio una versione di questo algoritmo eseguita contestualmente all'algoritmo di aggregazione dei profili, ovvero un unico algoritmo che effettui entrambe le operazioni. Bisogna però sottolineare che l'algoritmo di aggregazione dati impiega meno dell'uno per-cento del tempo di esecuzione totale dell'architettura, che è per la maggior parte imputabile alle comunicazioni tra i vari moduli. Al fine di affrontare appunto il problema delle comunicazioni fra moduli sono state cercate e testate diverse soluzioni che verranno mostrate nella prossima sezione.

2.4 Comunicazione fra moduli

In questa sezione verranno mostrate le soluzioni alternative alla comunicazione tramite Web Services che sono state prese in considerazione nell'ambito di questa tesi. In particolare si parlerà della comunicazione attraverso le Socket, della comunicazione mediante Remote Method Invocation e attraverso JNDI. In questa sezione, diversamente dalle precedenti ed al fine di non appesantire troppo la discussione, non ci soffermerà molto sull'implementazione tecnica delle diverse tecnologie di comunicazione, ma si cercherà di fornire una descrizione delle prestazioni misurate e delle differenze tra le diverse tecnologie.

2.4.1 Possibili alternative

Come evidenziato nel capitolo 1.2 le performance registrate dall'architettura, unite ad alcune considerazioni di fondo hanno fatto ritenere inadeguato l'utilizzo dei Web service per la comunicazione interna fra moduli dell'architettura. In particolare, è stata decisiva la lettura di due articoli scientifici ([6] e [7]) in cui viene mostrato come la serializzazione XML utilizzata dai Web service sia significativamente più lenta rispetto alla serializzazione binaria utilizzata da RMI. L'interoperabilità garantita dai Web service viene quindi pagata in termini di efficienza. Al fine, dunque, di valutare la possibilità di sostituire tale tecnologia di comunicazione con una più performante, sono state prese in esame diverse alternative. Ognuna di esse è stata prima studiata, poi implementata ed infine testata. Tutti i test sono stati eseguiti sul medesimo computer secondo le medesime modalità. Il risultato di questa operazione è quanto verrà esposto nelle prossime pagine.

RMI

La prima soluzione vagliata è stata la tecnologia per la programmazione distribuita RMI (Remote Method Invocation). RMI fornisce un'astrazione ad alto livello delle comunicazioni fra due oggetti Java. In particolare, utilizzando RMI, non è necessario implementare nessun tipo di programmazione di rete a basso livello (vedi 2.4.1), ma è possibile effettuare l'invocazione di metodi che verranno eseguiti in remoto, come se fossero eseguiti in locale. Il funzionamento percepito (non il funzionamento interno) della tecnologia RMI è il seguente. Le classi contenenti i metodi che si vuole rendere disponibili per essere invocati da remoto debbono registrarsi al sistema di naming di RMI. Tale sistema di naming viene realizzato attraverso un *RMI Registry*. L'*RMI Registry* implementa una mappa in cui ad ogni oggetto remoto è associata una stringa di caratteri che ne rappresenta il nome. È quindi possibile ottenere un riferimento ad un oggetto remoto effettuando un interrogazione dell'*RMI Registry*. Un Client, per poter effettuare un'invocazione di metodo da remoto ha inoltre bisogno di uno *STUB* della classe dell'oggetto remoto cui ci si riferisce. Tale *STUB* è generato in automatico da un'apposito programma. Senza tale *STUB* non è possibile effettuare alcuna invocazione.

Per studiarne le performance all'interno della nostra architettura sono stati realizzati un "server" RMI per ogni *Profile Manager* ed un "Client' per il *Context Provider*. Ogni server RMI è controllato da una Servlet. I test sono stati effettuati facendo eseguire, mediante una Servlet, l'*RMI Registry* all'interno del container JBoss, nella versione 3.2.4. Successivamente, sempre all'interno di JBoss, per ogni *Profile Manager* e attraverso altrettante Servlet sono stati registrati gli oggetti remoti che permettono di recuperare il profilo CC/PP di un utente. L'interrogazione è invece stata fatta da un Client esterno appositamente scritto per misurare i tempi di recupero dei profili ed eseguito all'interno della piattaforma di sviluppo Eclipse, nella versione 3.0.2. L'ambiente di esecuzione era il seguente: Pentium M 730 - 512MBytes Ram - Suse Linux 9.2 - KDE 3.4. Il tempo misurato per la prima richiesta al *Profile Manager OPM* è pari a $166 millisecondi$, così distribuiti:

1. 26.0*millisecondi* nella connessione all'RMI-Registry (operazione *LocateRegistry.getRegistry*);

2. 115.0*millisecondi* per recuperare un riferimento all'oggetto remoto (operazione *registry.lookup*);

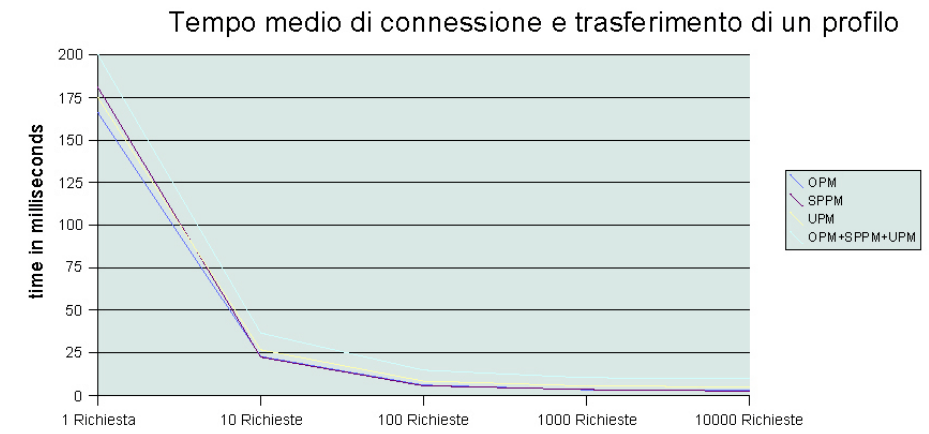
3. 25.0*millisecondi* per recuperare il profilo CC/PP mediante il metodo.

Come si può notare, la maggior parte del tempo è dovuto all'ottenimento del riferimento all'oggetto remoto. In figura 2.4.1 si possono vedere i tempi misurati al variare del numero di richieste. Nella prima tabella (e nel primo grafico) è mostrato il tempo medio per l'ottenimento del profilo, comprensivo di tutte le operazioni sopra-elencate. Nella seconda tabella (e nel secondo grafico) è mostrato il tempo medio della sola operazione di richiesta e ricezione del profilo, una volta che è stato ottenuto il riferimento all'oggetto remoto. Come si può notare i tempi misurati nel caso in cui sia ogni volta necessario ottenere un riferimento all'oggetto remoto sono molto più elevati rispetto a quelli necessari per il semplice recupero del profilo.

RMI

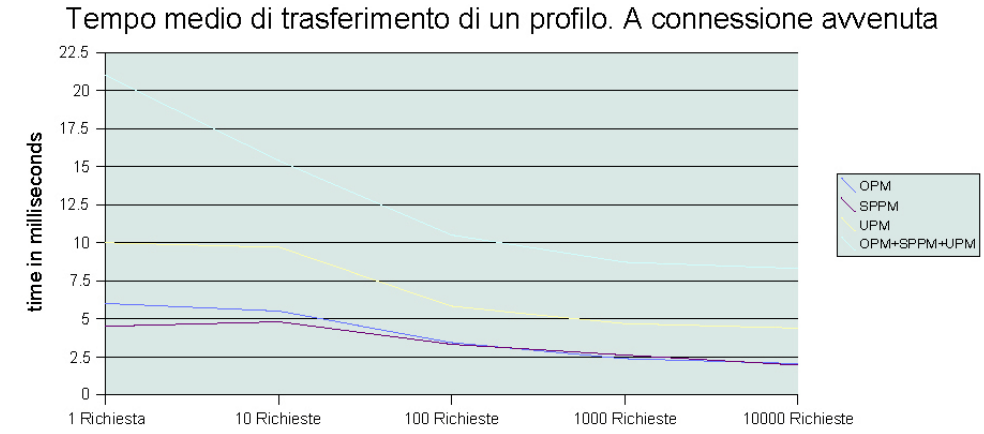
TEMPO MEDIO DI CONNESSIONE E TRASFERIMENTO DI UN PROFILO

	OPM	SPPM	UPM	OPM+SPPM+UPM	
1 Richiesta	166	181	175	200	*millisecondi*
10 Richieste	22.8	22.3	26.7	36.5	*millisecondi*
100 Richieste	5.96	5.6	8.26	14.78	*millisecondi*
1000 Richieste	3.17	3.37	5.41	10.24	*millisecondi*
10000 Richieste	2.6	2.37	4.89	9.74	*millisecondi*



TEMPO MEDIO DI TRASFERIMENTO DI UN PROFILO. A CONNESSIONE AVVENUTA.

	OPM	SPPM	UPM	OPM+SPPM+UPM	
1 Richiesta	6	4.5	10	21	*millisecondi*
10 Richieste	5.5	4.8	9.7	15.4	*millisecondi*
100 Richieste	3.43	3.31	5.83	10.5	*millisecondi*
1000 Richieste	2.38	2.61	4.68	8.71	*millisecondi*
10000 Richieste	2.05	1.95	4.39	8.3	*millisecondi*



Socket

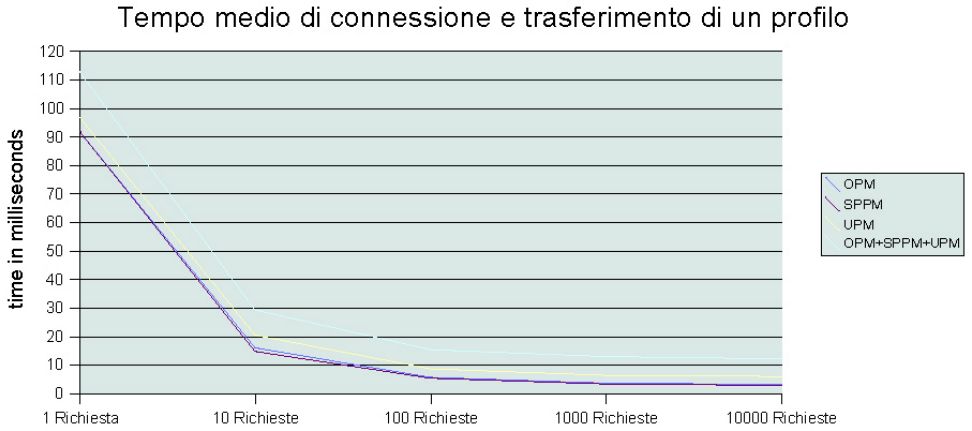
Ispirandosi all'articolo della Sun [8], la seconda soluzione che è stata vagliata è quella relativa appunto alle socket. Questa tecnologia è stata presa in considerazione perché, sempre secondo l'articolo [8], essa dovrebbe garantire prestazioni migliori rispetto ad RMI. Infatti RMI, offrendo una serie di facilitazioni ed astrazioni che permettono di semplificare la programmazione distribuita, risulterebbe meno efficiente rispetto alle Socket. Questa soluzione prevede quindi l'utilizzo di "normali" socket per la comunicazione fra moduli. La particolarità di questa soluzione sta nel fatto che le comunicazioni tra Client e Server sfruttano la serializzazione degli oggetti di Java. Tale serializzazione permette l'invio dell'oggetto della classe *ProfileAndPolicies* contenente il profilo in modo molto semplice. Per testare questa soluzione sono stati implementati un server multithreading per ogni *Profile Manager* ed un Client per il *Context Provider*. I codici sorgenti di Client e del Server sono un adattamento di quelli pubblicati dalla Sun nel sopracitato articolo [8].

L'ambiente e le modalità di esecuzione dei test sono le medesime adottate per RMI. In figura 2.4.1 si possono vedere i tempi misurati al variare del numero di richieste. Nella prima tabella (e nel primo grafico) sono mostrati i tempi per l'ottenimento del profilo. Nella seconda tabella (e nel primo grafico) sono mostrati i medesimi tempi, per richieste successive alla prima. Per effettuare le misurazioni relative alla seconda tabella, prima di iniziare a cronometrare l'esecuzione delle richieste, il Client incaricato di effettuare il testing eseguiva quindi una prima connessione ai relativi *Profile Manager*. Come si può notare il tempo per la prima connessione rimane comunque elevato, ma per ogni *Profile Manager* è comunque poco più della metà rispetto a quello richiesto da RMI. All'aumentare del numero di richieste invece le differenze si assottigliano fino a quando la situazione si inverte a vantaggio di RMI che esegue 10000 richieste in un tempo medio di $9.74 millisecondi$, contro i $12.26 millisecondi$ in media impiegati dalle Socket.

Socket

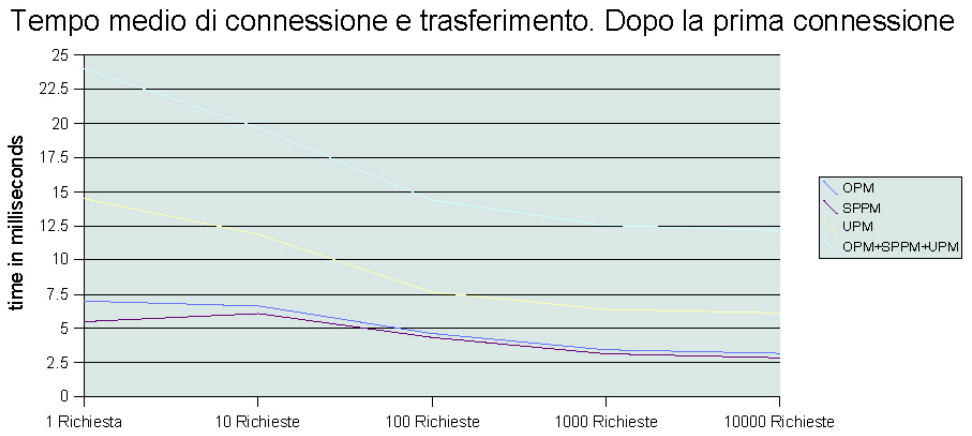
TEMPO MEDIO DI CONNESSIONE E TRASFERIMENTO DI UN PROFILO

	OPM	SPPM	UPM	OPM+SPPM+UPM	
1 Richiesta	92	92	97	113	*millisecondi*
10 Richieste	16.1	14.9	20.6	29.4	*millisecondi*
100 Richieste	5.81	5.46	8.75	15.47	*millisecondi*
1000 Richieste	3.73	3.45	6.52	13.07	*millisecondi*
10000 Richieste	3.32	2.97	6.14	12.26	*millisecondi*



TEMPO MEDIO DI CONNESSIONE E TRASFERIMENTO. DOPO LA PRIMA CONNESSIONE

	OPM	SPPM	UPM	OPM+SPPM+UPM	
1 Richiesta	7	5.5	14.5	24	*millisecondi*
10 Richieste	6.65	6.1	11.9	19.7	*millisecondi*
100 Richieste	4.64	4.34	7.68	14.39	*millisecondi*
1000 Richieste	3.45	3.15	6.41	12.55	*millisecondi*
10000 Richieste	3.19	2.86	6.14	12.16	*millisecondi*



JNDI

La terza (ed ultima) alternativa che è stata analizzata è quella rappresentata dalla comunicazione attraverso JNDI (Java Naming and Directory Interface). Come già detto nella sezione 2.2.1, JNDI è un interfaccia per l'accesso a sistemi di naming e directory che offre un livello di astrazione nei confronti dei sistemi di naming simile a quello offerto da JDBC nei confronti dei database relazionali. L'idea di testare anche questa tecnologia è nata in quanto JNDI è (nella piattaforma J2EE) uno degli standard di comunicazione fra Enterprise Java Bean. È sembrato naturale prendere in considerazione anche questa possibilità dato che attualmente i *Profile Manager* sono stati trasformati in delle applicazioni J2EE contenenti ognuna un EJB di sessione e dato che il *Contex Provider* era già stato realizzato con i medesimi criteri. Inoltre, rispetto ad RMI, JNDI ha il vantaggio di non richiedere che il Client abbia uno *STUB* della classe remota, ma è sufficiente che questo conosca l'interfaccia di tale classe.

Come precedentemente detto, ogni EJB ha automaticamente accesso ad uno speciale sistema di naming chiamato ENC (Environment Naming Context) gestito dal container degli EJB, il quale implementa e fornisce anche il provider per l'accesso a tale ENC. Nel nostro caso è stato utilizzato il provider per ENC fornito dal container JBoss. Per realizzare la comunicazione tramite JNDI, in ogni EJB associato ad un *Profile Manager* è stato inserito un metodo "business" avente il compito di recuperare e restituire il profilo CC/PP associato ad un utente. Come questo metodo realizzi le proprie funzionalità è spiegato nel capitolo 2.2.1. Il *Context Provider* è stato quindi modificato, inserendo una classe che si occupa di interrogare l'ENC, ottenere un riferimento all'EJB di ogni *Profile Manager* ed invocare quindi in remoto il metodo dell'EJB, per ottenere il profilo CC/PP. L'ambiente e le modalità di esecuzione dei test sono le medesime adottate per RMI e le Socket. In figura 2.4.1 sono mostrati i tempi misurati al variare del numero delle richieste. Nella prima tabella (e nel primo grafico) è mostrato il tempo medio per l'ottenimento del profilo, comprensivo di tutte le oper-

azioni sopra-citate. Nella seconda tabella (e nel secondo grafico) è mostrato il tempo medio della sola operazione di invocazione remota del metodo per recuperare il profilo, una volta che è stato ottenuto il riferimento all'oggetto remoto. Il tempo misurato per la prima richiesta al *Profile Manager OPM* è pari a 318 millisecondi, così distribuiti:

1. 55*millisecondi* per la creazione di un nuovo contesto iniziale tramite l'invocazione di *new InitialContext()*, durante questa fase viene contattato il provider JNDI sull'host e sulla porta desiderati;

2. 200*millisecondi* per l'ottenimento di un riferimento all'oggetto desiderato tramite l'invocazione del metodo *context.lookup*;

3. 7*millisecondi* per l'operazione di verifica della correttezza del casting dell'oggetto remoto, tramite l'invocazione del metodo *PortableRemoteObject.narrow*;

4. 31*millisecondi* per recuperare l'interfaccia remota dell'EJB tramite l'operazione *ejb.create()*;

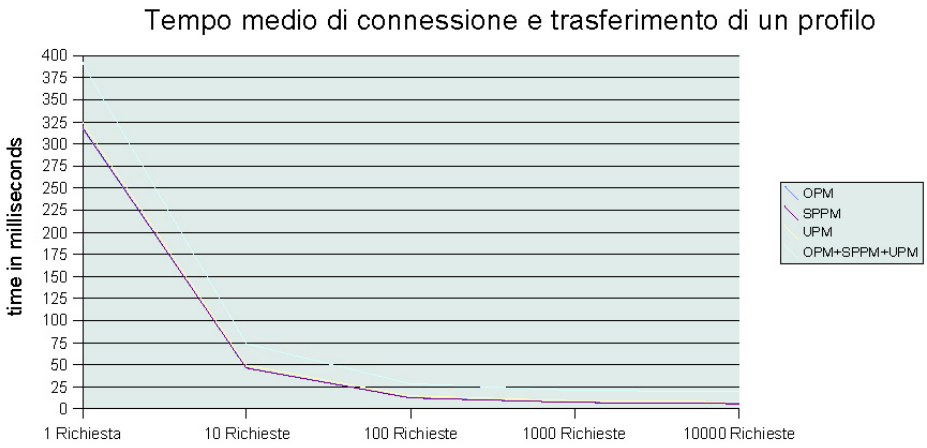
5. 25*millisecondi* per recuperare il profilo CC/PP tramite l'invocazione del metodo *getprofile* sull'interfaccia remota.

Anche in questo caso il tempo medio per il recupero dei profili scende vertiginosamente già dalle primissime richieste successive alla prima, tuttavia non scende mai ai livelli della comunicazione tramite Socket o RMI.

JNDI

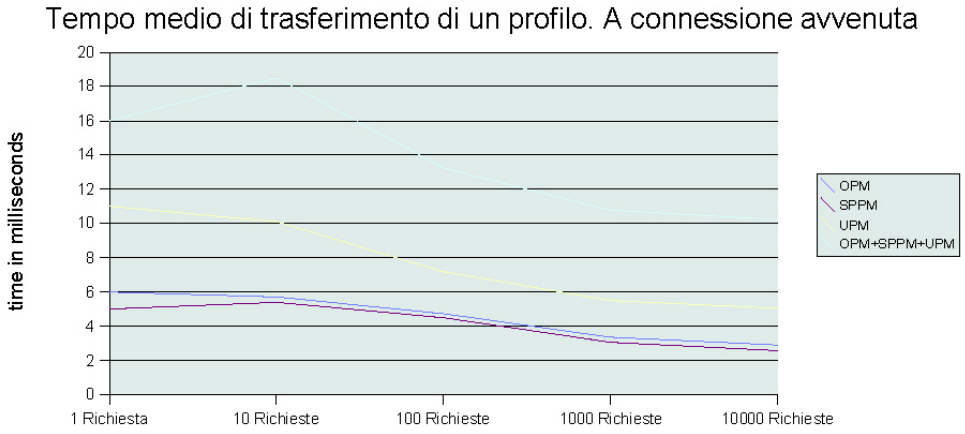
TEMPO MEDIO DI CONNESSIONE E TRASFERIMENTO DI UN PROFILO

	OPM	SPPM	UPM	OPM+SPPM+UPM	
1 Richiesta	318	319	325	392	*millisecondi*
10 Richieste	46.6	46.3	49.7	73.8	*millisecondi*
100 Richieste	12.84	12.46	14.88	28.75	*millisecondi*
1000 Richieste	7.33	7.46	9.47	21.06	*millisecondi*
10000 Richieste	6.14	5.98	8.3	19.81	*millisecondi*



TEMPO MEDIO DI TRASFERIMENTO DI UN PROFILO. A CONNESSIONE AVVENUTA.

	OPM	SPPM	UPM	OPM+SPPM+UPM	
1 Richiesta	6	5	11	16	*millisecondi*
10 Richieste	5.7	5.4	10.1	18.5	*millisecondi*
100 Richieste	4.73	4.49	7.2	13.24	*millisecondi*
1000 Richieste	3.36	3.06	5.5	10.78	*millisecondi*
10000 Richieste	2.91	2.57	5.06	10.23	*millisecondi*



2.4.2 Conclusioni

Le suddette misurazioni non sembrano lasciare adito a dubbi riguardo a quale sia la tecnologia di comunicazione più efficiente: la comunicazione tramite Socket risulta la più performante e ciò è molto probabilmente dovuto al fatto che non vi sono operazioni da effettuare all'infuori della richiesta del profilo, mentre, sia in RMI, che in JNDI vi è un sistema di naming con cui bisogna interfacciarsi prima di poter effettuare la richiesta vera e propria. Confrontando in particolare la distribuzione del tempo di esecuzione di una singola richiesta effettuata con RMI e JNDI è possibile notare come l'operazione più costosa in termini di tempo sia quella per l'ottenimento del riferimento all'oggetto remoto, ovvero l'operazione di *lookup*.

Tuttavia, i test i cui risultati sono pubblicati in queste pagine sono stati effettuati utilizzando un Client esterno all'architettura, eseguito oltretutto separatamente rispetto al container JBoss. Risulta quindi necessario, prima di poter giungere a delle conclusioni, effettuare dei test approfonditi delle prestazioni misurate eseguendo il Client che si occupa di effettuare il testing, all'interno di JBoss o, meglio ancora, registrando direttamente i tempi per recuperare i profili, misurati dal Context Provider. Questa operazione è già in parte stata effettuata ma non in modo sufficiente approfondito da poterne pubblicare i risultati. Le suddette considerazioni valgono soprattutto per quanto riguarda la tecnologia JNDI. Nell'architettura presentata, infatti, la comunicazione avviene tra due EJB (*Context Provider* e *Profile Manager*). Sarebbe quindi necessario eseguire un'analisi approfondita delle performance della tecnologia JNDI, utilizzandola completamente all'interno di un EJB container quale JBoss, prima di poter giungere ad una decisione. In definitiva, non è quindi possibile in questa sede giungere ad una decisione finale riguardo a quale sia la tecnologia di comunicazione da adottare. Questo lavoro può però essere considerato un primo passo verso tale scelta. Infatti, avendo implementato tutte e tre le tecnologie sopracitate è possibile testarle a fondo. Inoltre si potrebbe anche pensare di scegliere di utilizzarne più di una scegliendo di volta in volta quella che si ritiene più opportuna per le

specifiche necessità del momento.

Comparazione delle performance

TEMPO MEDIO DI CONNESSIONE E TRASFERIMENTO DI UN PROFILO

Socket

	OPM	SPPM	UPM	OPM+SPPM+UPM	
1 Richiesta	92	92	97	113	*ms*
10 Richieste	16.1	14.9	20.6	29.4	*ms*
100 Richieste	5.81	5.46	8.75	15.47	*ms*
1000 Richieste	3.73	3.45	6.52	13.07	*ms*
10000 Richieste	3.32	2.97	6.14	12.26	*ms*

RMI

	OPM	SPPM	UPM	OPM+SPPM+UPM	
1 Richiesta	166	181	175	200	*ms*
10 Richieste	22.8	22.3	26.7	36.5	*ms*
100 Richieste	5.96	5.6	8.26	14.78	*ms*
1000 Richieste	3.17	3.37	5.41	10.24	*ms*
10000 Richieste	2.6	2.37	4.89	9.74	*ms*

JNDI

	OPM	SPPM	UPM	OPM+SPPM+UPM	
1 Richiesta	318	319	325	392	*ms*
10 Richieste	46.6	46.3	49.7	73.8	*ms*
100 Richieste	12.84	12.46	14.88	28.75	*ms*
1000 Richieste	7.33	7.46	9.47	21.06	*ms*
10000 Richieste	6.14	5.98	8.3	19.81	*ms*

TEMPO MEDIO DI TRASFERIMENTO DI UN PROFILO. A CONNESSIONE AVVENUTA.

RMI

	OPM	SPPM	UPM	OPM+SPPM+UPM	
1 Richiesta	6	4.5	10	21	*ms*
10 Richieste	5.5	4.8	9.7	15.4	*ms*
100 Richieste	3.43	3.31	5.83	10.5	*ms*
1000 Richieste	2.38	2.61	4.68	8.71	*ms*
10000 Richieste	2.05	1.95	4.39	8.3	*ms*

JNDI

	OPM	SPPM	UPM	OPM+SPPM+UPM	
1 Request	6	5	11	16	*ms*
10 Requests	5.7	5.4	10.1	18.5	*ms*
100 Requests	4.73	4.49	7.2	13.24	*ms*
1000 Requests	3.36	3.06	5.5	10.78	*ms*
10000 Requests	2.91	2.57	5.06	10.23	*ms*

Capitolo 3

Conclusioni e sviluppi futuri

L'attività di reingegnerizzazione svolta ha portato ad un significativo miglioramento delle prestazioni globali dell'architettura *CARE*. In particolare, il tempo di aggregazione dei profili è adesso circa 1/30 del tempo necessario precedentemente a questa tesi mentre, utilizzando le Socket per la comunicazione tra moduli, il tempo totale necessario per servire una richiesta è stato ridotto ad 1/4 del tempo precedentemente richiesto. La possibilità di gestire i profili CC/PP tramite una libreria ad-hoc, la progettazione ed implementazione di un algoritmo efficiente per l'aggregazione dei profili stessi, la reingegnerizzazione dei moduli *Profile Manager* e delle comunicazioni tra questi ed il *Context Provider* forniscono non solo un significativo miglioramento delle prestazioni, ma anche una miglioramento della manutenibilità del codice dell'architettura *CARE*. Il modulo di aggregazione dei dati di contesto permette ora di conoscere da quale profile manager proviene ogni singolo valore del profilo aggregato, mentre l'approccio per componenti con cui sono stati reingegnerizzati i *Profile Manager* ne garantisce la massima scalabilità ed espandibilità. Infine, la possibilità di scegliere tra diverse tecnologie di comunicazione incrementa la versatilità dell'architettura.

Essendo stata reingegnerizzata tutta la logica necessaria all'aggregazione dei profili utente, partendo dai database dei *Profile Manager* fino al modulo *Inference Engine* (escluso), il lavoro effettuato si configura come completo

e autonomo. Tuttavia vi sono alcuni aspetti che debbono ancora di essere approfonditi o implementati:

Comunicazione fra moduli Come precedentemente detto, lo studio effettuato sulle possibili tecnologie di comunicazione tra il *Context Provider* ed i *Profile Manager* non ha ancora portato ad una decisione definitiva su quale sia la tecnologia migliore per le esigenze attuali di questa architettura. È necessario quindi approfondire tale studio effettuando un profiling delle prestazioni delle varie tecnologie di comunicazione, utilizzate all'interno del container J2EE attualmente adottato.

Caching L'architettura attuale non è ancora dotata di meccanismi di caching dei dati di contesto. Anche se le prestazioni dell'architettura sono state notevolmente incrementate, l'utilizzo di politiche di caching è indispensabile per rendere l'architettura utilizzabile in applicazioni reali. Una implementazione completa del meccanismo dei trigger permetterebbe di implementare efficacemente il caching.

Gestione privacy Un'esigenza molto sentita è quella relativa alla gestione della privacy degli utenti i cui profili, essendo potenzialmente comprensivi di dati sensibili, debbono essere protetti da opportune misure di sicurezza e debbono poter essere gestiti e controllati dagli utenti stessi.

Personalmente ritengo che il lavoro svolto per questa tesi durante il periodo di stage presso il laboratorio DaKWE (Data Knowledge and Web Engineering) abbia portato ad una crescita notevole delle mie conoscenze. Ho imparato molto sul linguaggio di programmazione Java, sulle sue strutture dati e sulle problematiche relative all'implementazione efficiente di algoritmi di aggregazione dati. Durante questa ricerca ho inoltre potuto sperimentare e confrontare diverse tecnologie per la programmazione distribuita ed, infine, mi ha particolarmente entusiasmato la parte relativa alla reingegnerizzazione dei *Profile Manager*, in quanto mi ha permesso di avvicinarmi al mondo della progettazione per componenti di applicazioni J2EE.

Sono quindi soddisfatto dell'esperienza acquisita e dei risultati raggiunti. Credo fermamente nella scienza come metodologia disciplinata e filtrata per migliorare le condizioni di vita degli uomini e, anche se mi rendo perfettamente conto che essa non sia ne fonte di verità assoluta ne di perfezione, ritengo che essa, insieme alle arti e alla filosofia, costituisce quanto di più alto ed elevato l'uomo abbia mai realizzato. Non posso che ritenermi quindi onorato di aver potuto collaborare con il mondo della ricerca accademica e di aver potuto offrire un mio seppur modesto contributo.

Elenco delle figure

Bibliografia

[1] A. Agostini, C. Bettini, N. Cesa-Bianchi, D. Maggiorini, D. Riboni, M. Ruberl, C. Sala, and D. Vitali. Towards Highly Adaptive Services for Mobile Computing. In *Proceedings of IFIP TC8 Working Conference on Mobile Information Systems (MOBIS)*, 2004.

[2] H. Boley, S. Tabet, and G. Wagner. Design Rationale of RuleML: A Markup Language for Semantic Web Rules. In *Proceedings of the International Semantic Web Working Symposium (SWWS)*, pages 381–401, 2001.

[3] S. Ceri, G. Gottlob, and L. Tanca. *Logic Programming and Databases*. Springer-Verlag, 1990.

[4] G. Klyne, F. Reynolds, C. Woodrow, H. Ohto, J. Hjelm, M. H. Butler, and L. Tran. Composite Capability/Preference Profiles (CC/PP): Structure and Vocabularies 1.0. W3C Recommendation, W3C, January 2004. http://www.w3.org/TR/2004/REC-CCPP-struct-vocab-20040115/.

[5] C. Bettini and D. Riboni. Profile Aggregation and Policy Evaluation for Adaptive Internet Services. In *Proceedings of The First Annual International Conference on Mobile and Ubiquitous Systems: Networking and Services (Mobiquitous)*, pages 290–298. IEEE, 2004.

[6] *Java RMI, RMI Tunneling and Web Services Comparison and Performance Analysis*
http://portal.acm.org/citation.cfm?id=997146

[7] *Object Serialization Analysis and Comparison in Java and .NET*
http://portal.acm.org/citation.cfm?id=944589

[8] *Advanced Socket Programming*
http://java.sun.com/developer/technicalArticles/ALT/sockets/

[9] Gamma et al. : *Design Patterns: Elements of Reusable Object-Oriented Software*

[10] *Java 2 Platform Enterprise Edition Specification, v1.4*
http://java.sun.com/j2ee/j2ee-1_4-fr-spec.pdf

[11] *Enterprise JavaBeans Fundamentals*
http://java.sun.com/developer/onlineTraining/EJBIntro/EJBIntro.html

[12] *Java Naming and Directory Interface API*
http://www.mokabyte.it/2002/09/jndi-1.htm