Consiglio Nazionale delle Ricerche
Istituto di Calcolo e Reti ad Alte Prestazioni

JAebXR

Java API for ebXML Registries

A. Messina, P. Storniolo

Rapporto Tecnico N.:
RT-ICAR-PA-13-02 **Dicembre 2013**

Consiglio Nazionale delle Ricerche, Istituto di Calcolo e Reti ad Alte Prestazioni (ICAR)
– Sede di Cosenza, Via P. Bucci 41C, 87036 Rende, Italy, URL: *www.icar.cnr.it*
– Sede di Napoli, Via P. Castellino 111, 80131 Napoli, URL: *www.na.icar.cnr.it*
– Sede di Palermo, Viale delle Scienze ed.11, 90128 Palermo, URL: *www.pa.icar.cnr.it*

Consiglio Nazionale delle Ricerche
Istituto di Calcolo e Reti ad Alte Prestazioni

JAebXR

Java API for ebXML Registries

A. Messina[1], P. Storniolo[1]

Rapporto Tecnico N.:
RT-ICAR-PA-13-02 **Dicembre 2013**

---

[1] Istituto di Calcolo e Reti ad Alte Prestazioni, ICAR-CNR, Sede di Palermo, Viale delle Scienze edificio 11, 90128 Palermo.

Sommario

1 Introduzione

Le attività di sviluppo nell'ambito della realizzazione del Fascicolo Sanitario Elettronico di Seconda Generazione (FSE2) hanno evidenziato tutta una serie di peculiarità ed idiosincrasie dovute alle modalità tipicamente utilizzate nelle interazioni con i registri ebXML.

Sia con la prima implementazione di registro presa a riferimento *OMAR v3.1*, che con la sua evoluzione rappresentata dal fork *eRIC v3.2*, al fine di realizzare uno strato di middleware quanto più indipendente dalla particolare implementazione del registro, l'unica possibilità è quella di utilizzare le interfacce applicative fornite dalle componenti di provider JAXR messe a disposizione dai registri stessi.

Tale substrato non fa altro che comportarsi da gateway ebXML-JAXR: si utilizza la tecnologia JAXR con le relative classi di oggetti *InfoModel* per le interazioni con i client del registro, mentre internamente si utilizza esclusivamente ebXML con le classi di oggetti ebXML RIM definiti dallo standard OASIS RegRep 3.0.

Visto che le specifiche di FSE richiedono l'utilizzo di oggetti ebXML RIM e che le interazioni con i registri avvengono in JAXR, ne consegue una necessaria doppia conversione dei dati oggetto delle transazioni:

1. da ebXML RIM a JAXR InfoModel nel livello client/middleware;
2. da JAXR InfoModel a nuovamente ebXML RIM nel livello di gateway JAXR del registro.

Si è quindi ritenuto opportuno sviluppare una API per realizzare client di registri ebXML che lavorassero direttamente con oggetti di tipo ebXML RIM piuttosto che JAXR.

La libreria *Java API for ebXML Registries* (JAebXR) qui presentata è da ausilio proprio in questa attività, estendendo le funzionalità delle API JAXR mantenendone peculiarità ed architettura e prefigurandosi anche come *JAXR provider di livello 1*, poiché consente l'interfacciamento sia a registri ebXML che a registri di tipo UDDI.

2 Architettura di ebXML

2.1 Introduzione

ebXML (Electronic Business Extensible Markup Language) è un progetto internazionale ideato da UN/CEFACT (United Nation Center for Trade Facilitation and Electronic Business) e da OASIS (Organization for the Advancement of Structured Information Standard).

L'obiettivo del progetto è quello di fornire un framework che permetta al linguaggio XML di essere utilizzato in modo uniforme e consistente nello scambio di messaggi tra applicativi e utenti in un contesto di commercio elettronico.

Il vantaggio fornito da ebXML è apprezzabile anche per aziende di piccola e media grandezza e che abbiano un livello di informatizzazione anche minimo; la possibilità che viene fornita da ebXML è uno scambio di messaggi basati su XML che favorisca il commercio elettronico della ditta con altre strutture (della stessa società o di ditte diverse) geograficamente localizzate in luoghi diversi.

Questo vantaggio è garantito da un vocabolario di termini comuni che permette una comunicazione commerciale con messaggistica di tipo standard.

Una delle caratteristiche di ebXML è quella di fornire contatti d'affari *ad hoc*.

Vediamo un esempio per chiarificare il concetto.

Supponiamo di comperare ortaggi da un supermarket A. Dopo diverso tempo abbiamo sviluppato una relazione d'affari con il supermarket, basato sui prodotti forniti e sul processo intrapreso per comprarli, come l'interazione con i commessi e il pagamento tramite carta di credito dei beni. Possiamo definire questa relazione *ad hoc*.

Supponiamo ora che entri nel mercato un nuovo supermarket B, che venda lo stesso prodotto a prezzi inferiori. Sarebbe nel nostro interesse intraprendere degli affari con questo nuovo supermarket, e possiamo farlo perché ci aspettiamo di non incontrare ostacoli: i commessi si comporteranno come nel primo supermercato (parlano la nostra stessa lingua) e possiamo pagare la merce con la carta di credito.

Il commercio elettronico, in realtà, ha costi di infrastrutture che influiscono sul prezzo totale dell'affare. Durante una transazione elettronica, per esempio, i partecipanti devono affrontare i costi del software, del trasporto dati, così come decidere le politiche di interazione e sicurezza.

In pratica, se un nuovo fornitore offrisse beni a minor prezzo, potrebbe accadere di dover considerare i costi del servizio di acquisto come elemento discriminante. Ad un minor costo del prodotto potrebbe infatti corrispondere un maggior costo o complicazione del processo di acquisto.

Tornando all'esempio precedente, se il supermercato B avesse solo commessi stranieri, che parlassero esclusivamente una lingua diversa dalla nostra e non venissero accettati pagamenti con carta di credito ma solo con contanti e di piccolo taglio, è facile comprendere che il risparmio ottenuto sul bene acquistato andrebbe perso nell'adeguamento alle condizioni di vendita imposte.

Uno dei valori basilari di ebXML è la possibilità di ubiquità da una prospettiva tecnologica. E' costruito su *XML*, *SOAP*, *HTTP* e *SMTP*, tutti standard aperti e senza barriere. In teoria, mettere al centro l'ubiquità della tecnologia, dovrebbe permettere al commercio elettronico un approccio *ad hoc* al concetto di libero mercato che abbiamo visto nell'esempio del supermercato.

2.2 Elementi di ebXML

Gli elementi che compongono la struttura di ebXML sono molteplici:

- *Registry*: un server centrale che contiene una varietà di dati necessari per il funzionamento di ebXML. Tra le varie informazioni che il *Registry* mette a disposizione di XML troviamo: *Business Process & Information Meta Models*, *Core Library*, *Collaboration Protocol Profiles*, e *Business Library*. Il Registry in ebXML è necessario per individuare un partner adatto e per ricavare informazioni riguardo agli elementi necessari per rapportarsi con quel partner;

- *Business Processes*: attività che un'impresa può intraprendere (e per la quale sta cercando uno o più partner). Un Business Process è formalmente descritto da uno Schema di Specifiche (Business Process Specification Schema: uno Schema XML e un DTD), ma potrebbe essere modellato anche in UML (Unified Modeling Language);

- *Collaboration Protocol Profile (CPP)*: profilo archiviato in un *Registry* da un'impresa che vorrebbe intraprendere delle transizioni ebXML. Il CPP specifica alcuni Business Process dell'impresa e anche alcune Business Service Interface supportate;

- *Business Service Interface*: i modi in cui un'impresa è in grado di portare avanti le transazioni necessarie nei suoi Business Process. La Business Service Interface include inoltre i tipi di Business Message supportati e i protocolli sui quali questi messaggi dovranno viaggiare;

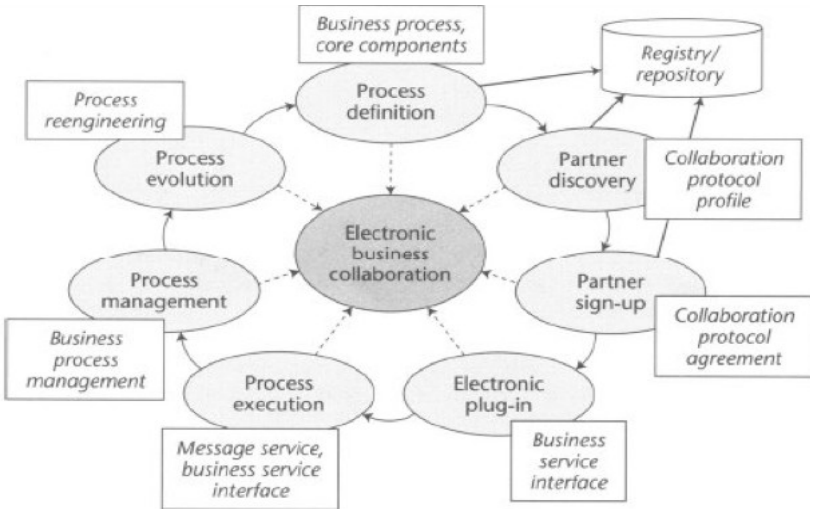
- *Business Message*: le informazioni correnti comunicate come parte della transazioni d'affari. Un messaggio contenente diversi strati. Al livello più esterno deve essere utilizzato in protocollo di comunicazione (tipo HTTP o SMTP). SOAP viene raccomandato come envelope (busta, strato esterno) per un messaggio "payload". Gli altri strati possono concernere criptazione o autenticazione;

- *Core Library*: un insieme di "parti" standard che possono essere utilizzate in elementi ebXML più grandi. Per esempio i Core Process possono essere referenziati dai Business Process. Alla Core Library contribuisce l'iniziativa ebXML stessa, mentre agli elementi più grandi possono contribuire specifiche imprese o industrie;

- *Collaboration Protocol Agreement (CPA)*: in pratica, un contratto tra due o più imprese che può essere derivato automaticamente dai CPP delle rispettive compagnie. Per esempio se un CPP afferma "Io *posso* fare X", un CPA afferma "Noi *faremo* X insieme";

- *Simple Object Access Protocol (SOAP)*: un protocollo di W3C per lo scambio di informazioni in un ambiente distribuito sottoscritto dall'iniziativa ebXML.

La parte di SOAP interessante per ebXML è la funzione di envelope che definisce un framework per la descrizione di cosa contiene il messaggio e come processarlo.



2.3 Funzionamento di ebXML

Per ottenere un approccio ad hoc, come visto precedentemente, ebXML fornisce un framework completo per le interazioni d'affari, le cui informazioni vengono inviate come un insieme di specifiche indipendenti dal partecipante all'affare. Il framework cerca di soddisfare i seguenti punti:

- Descrivere il processo e le interfacce specifiche;
- Condividere il processo con altri partner;
- Conoscere quali processi può supportare il partner;
- Descrivere i messaggi per una particolare transizione;
- Descrivere la politica di sicurezza e la configurazione tecnica da usare.

Molte di queste informazioni possono essere salvate in un registro condiviso che ha lo scopo di rendere centralizzati gli accordi dell'affare e i processi, l'*ebXML registry*.

Accordarsi per una nuova relazione d'affari significa accedere al registry di ebXML che, solitamente, viene gestita dalla fase di funzionamento corrente.

L'architettura di base prevede tre fasi:

1. fase di implementazione;
2. fase di ricerca;
3. fase di esecuzione.

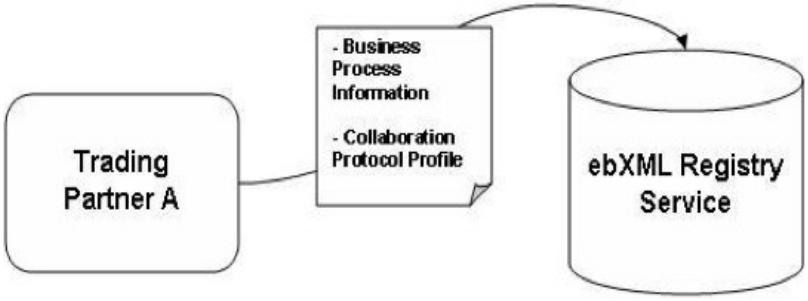
Per ogni fase sono definiti processi e meccanismi di sicurezza propri. In generale si possono immaginare le prime due fasi come un'ipotetica stretta di mano tra i due contraenti l'affare, mentre la terza come la reale attuazione dello scambio commerciale.

2.3.1 Fase di implementazione

Inizia con la decisione dei partner di svolgere una transizione d'affari tramite il framework ebXML. Il partner analizza i suoi processi e li pubblica nel registry preoccupandosi di renderli compatibili con la generalizzazione fornita da ebXML.

In questa fase deve essere prodotta un'implementazione, o dalle specifiche del nucleo di ebXML stesso o da terzi produttori.

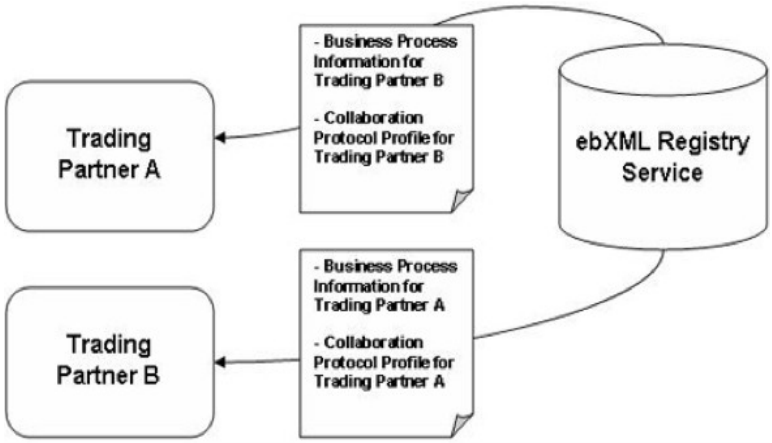
Come risultato si ottiene un framework funzionante contenente un insieme di processi e interfacce. Il *Collaboration Protocol Profile* (CPP) viene creato in questo momento.



2.3.2 Fase di ricerca

In questa fase i partner consultano il *registry* per vedere (scoprire) i processi e le interfacce pubblicate dall'altro partner. Tipicamente i CPP di un partner specifico vengono scambiati in questa fase.

Il CPP descrive i dettagli dei processi, compresi quelli riguardanti la sicurezza, il trasporto e l'affidabilità.

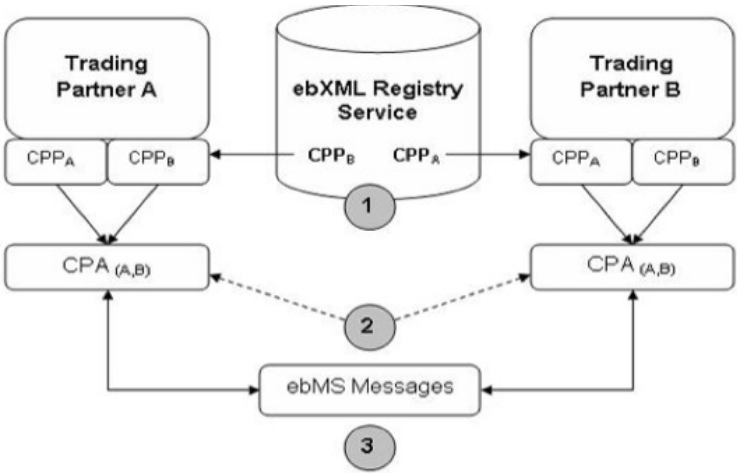


2.3.3 Fase di esecuzione

Gli accessi al registry sono terminati nelle fasi precedenti, per cui in questa avvengono esclusivamente gli scambi di messaggi tra i contraenti.

Le istanze di CPP pubblicate da ciascuno vengono ristrette per formare un Accordo sul *Collaboration Protocol Agreement* (CPA). Questo è un particolare accordo legato alle richieste formulate dai singoli CPP.

rima di iniziare lo scambio dei messaggi con ebMS, solitamente i due contraenti si preoccupano di verificare la consistenza del CPA creato: quando si verifica da entrambi i lati la consistenza, comincia lo scambio di messaggi.



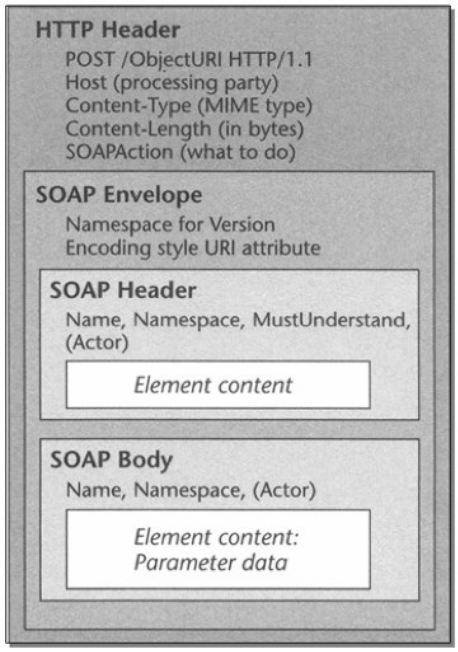
Facendo riferimento alla figura precedente possiamo riassumere la fase di runtime in 3 passi:

- Ogni partner è responsabile di ottenere i documenti CPP necessari per accordarsi con gli altri partner. Queste informazioni vengono ottenute tramite l'accesso ad un Registry;
- Con i CPP ottenuti vengono creati i CPA, che esplicitano la gamma di scelte offerte nei CPP;
- Sotto il controllo del CPA, i partner cominciano le transazioni tramite ebMS.

2.4 Formato dei messaggi ebXML

Il servizio incaricato della spedizione dei messaggi (ebMS) definisce una serie di elementi che verranno inseriti all'interno dell'intestazione (Header) o del corpo (Body) che soddisfano le specifiche SOAP per rendere il messaggio compatibile con le specifiche definite da ebXML.

Per ottenere ciò si dovranno incapsulare questi elementi in un multipart MIME che permetterà di inserire degli allegati nello stesso messaggio SOAP:



dove:

- **HTTP Header**: Rappresenta il contenitore di tutti il messaggio e specifica i protocolli di trasferimento;
- **SOAP Envelope**: Rappresenta il contenitore del messaggio;
- **SOAP Header**: è un elemento opzionale che contiene informazioni globali sul messaggio; ad esempio, nell'header potrebbe essere specificata la lingua di riferimento del messaggio, la data dell'invio, ecc.;
- **SOAP Body**: Rappresenta il contenuto vero e proprio del messaggio.

I principali obiettivi di ebXML a livello di messaging layer sono:

- Facilitare l’interscambio di messaggi di business all’interno di un framework XML;
- Essere neutrale rispetto al protocollo usato;
- Essere neutrale rispetto al contenuto del messaggio;
- Essere sicuro;
- Poter recapitare i messaggi in maniera affidabile.

Basandosi su questi obiettivi, ebXML ha prodotto la *ebXML Message Service Specification* che definisce uno standard per il trasporto sicuro ed affidabile dei messaggi.

I messaggi sono caratterizzati dai seguenti punti:

- Il loro formato è scritto in XML (è adottato il formato “SOAP with Attachments”);
- Non viene imposto alcun protocollo di trasporto da utilizzare;
- I Messaggi di Business, identificati dal corpo (payload) dei messaggi ebXML, non sono necessariamente espressi in XML. I messaggi XML-based sono trasportati dall’ebMS.

2.5 Registry e Repository

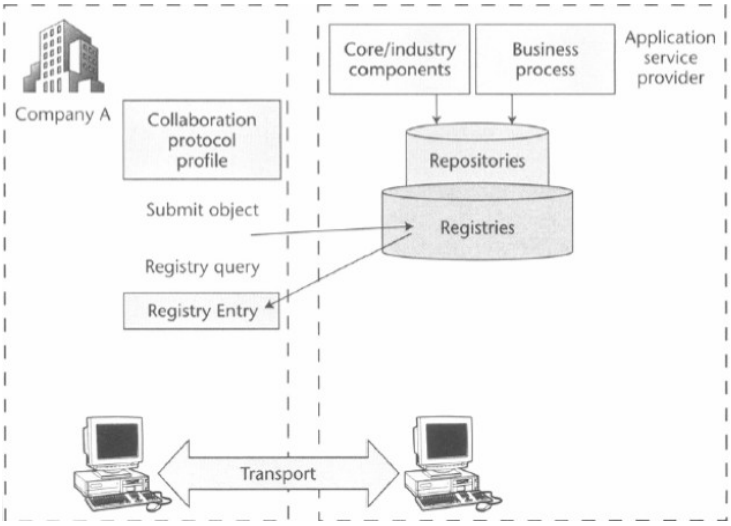
Il *Registry* fornisce una locazione stabile in cui rendere persistenti le informazioni ivi depositate da un apposito organismo. Tali informazioni vengono usate per facilitare partnership e transazioni Business to Business (B2B) basate su ebXML.

Il materiale sottoposto può consistere di schemi e documenti XML, descrizioni di processi, *ebXML Core Component*, descrizioni di contesto, modelli UML, informazioni varie circa le parti commerciali e persino componenti software.

Nel *Registry* vengono memorizzate le informazioni degli oggetti che sono depositati nel *Repository*. I due insieme possono essere considerati un database. Visti separatamente, il *Repository* contiene i modelli dei processi, i *core components*, i messaggi predefiniti e tutte quelle informazioni utili per permettere alle parti di scambiarsi informazioni in via elettronica.

Il *Registry* serve a collegare il *Repository* al mondo esterno; contiene quindi il software necessario per far in modo che i partner possano accedere alle informazioni presenti nel *Repository*.

Gli oggetti presenti nel *Repository* possono essere creati, aggiornati, cancellati solo con richieste fatte attraverso il *Registry*.



2.6 Core components

Per garantire l'interoperabilità fra diverse organizzazioni è necessario definire e creare una semantica di business comune (*Common Business Semantics*). L'ebXML ha quindi cercato di descrivere e specificare un nuovo approccio per risolvere il già noto problema dell'interoperabilità delle informazioni nel panorama e-business.

Tradizionalmente, gli standard per lo scambio di dati di business si sono sempre focalizzati su definizioni statiche dei messaggi, le quali non hanno però apportato un sufficiente grado di interoperabilità e flessibilità. Si è reso quindi necessario un nuovo e più flessibile metodo di standardizzazione.

Viene presentata una metodologia per sviluppare un insieme comune di componenti semantici elementari che rappresentano i tipi generali di dati di business correntemente in uso.

Inoltre viene definito un modo per creare nuovi vocabolari di business e per ristrutturare quelli già esistenti. Si cerca di garantire rappresentazioni delle

informazioni che siano allo stesso tempo leggibili dall'utente e processabili da sistemi automatici.

L'approccio scelto è flessibile in quanto la standardizzazione viene portata avanti secondo un approccio "syntax-neutral", indipendente dalla sintassi.

Usare i *Core Component* come parte fondamentale del framework ebXML aiuta a garantire che due diversi partner commerciali che usano differenti sintassi, utilizzino la semantica di business allo stesso modo, a condizione che le due sintassi siano basate sugli stessi *Core Component*.

Ciò permette un mapping semplice e "pulito" fra diversi tipi di definizioni di messaggi attraverso sintassi, confini industriali e geografici.

2.7 Collaboration Protocol Profile (CPP)

Lo scambio di informazioni fra due controparti richiede che ciascuna di esse sia a conoscenza delle collaborazioni binarie o multiparte (*Business Collaboration*) supportate dall'altra parte, quali il ruolo da questa ricoperto all'interno della collaborazione ed i dettagli tecnologici circa il modo in cui tale parte invia e riceve i messaggi. In alcuni casi, è necessario che le due controparti raggiungano un accordo su tali dettagli.

Per facilitare questo scambio di informazioni, ebXML descrive il *Collaboration Protocol Profile* (CPP) ed il *Collaboration Protocol Agreement* (CPA). Queste due tecnologie sono descritte in un'unica specifica.

Il CPP definisce le potenzialità ed i modi in cui una parte può impegnarsi in business elettronico con altre parti. Queste potenzialità sono sia tecnologiche (protocolli di messaggistica e di comunicazione supportati) sia commerciali (quali quelli supportati dalla *Business Collaboration*).

Una parte può descrivere se stessa in un unico CPP. Oppure, può creare differenti CPP che descrivono le *Business Collaboration* supportate, le sue operazioni in varie regioni del mondo, o le diverse sezioni della propria organizzazione.

Per facilitare la ricerca di possibili Business Partner, i CPP sono immagazzinati in un apposito repository pubblico quale l'ebXML Registry. Tramite un processo di ricerca ed individuazione, codificato nelle specifiche del repository, una parte commerciale può trovare adeguati Business Partner.

2.8 Collaboration Protocol Agreement (CPA)

Il *Collaboration Protocol Agreement* (CPA) definisce le potenzialità messe a disposizione dalle due controparti, sulle quali esse devono trovarsi in accordo prima di impegnarsi nelle transazioni descritte dal CPA.

Il CPA è indipendente dai particolari processi interni attuati da ciascuna parte. Questi processi interni vengono interfacciati verso l'esterno con le *Business Collaboration* descritte dal CPA e dal documento di Process-Specification (descritto nel Business Process Specification Schema - BPSS).

In tal modo non vengono esposti alla controparte dettagli dei processi interni.

L'intento del CPA è di fornire una specifica ad alto livello che può essere facilmente comprensibile ed allo stesso tempo rigorosa e precisa da poter essere implementata in maniera automatica.

L'informazione di un CPA è usata per configurare i sistemi dei due partner commerciali per rendere possibile lo scambio di messaggi durante lo svolgimento delle *Business Collaboration* selezionate.

I CPA non sono documenti cartacei bensì elettronici, che possono essere processati direttamente dai sistemi delle controparti per preparare ed eseguire gli scambi di informazioni voluti. Eventuali termini e condizioni legali di un accordo commerciale

sono al di là degli scopi di queste specifiche e quindi non sono inclusi né nel CPP né nel CPA.

Il CPA è creato mediante elaborazioni e negoziazioni derivanti dall’incrocio di due CPP. Ad esempio: un CPA può includere solamente quegli elementi che sono comuni o compatibili fra le due parti.

3 OASIS Regrep 3.0

3.1 Introduzione

Un Registro ebXML è un sistema informativo in grado di gestire in sicurezza qualunque tipo di contenuto digitale (*RepositoryItem*) e i metadati standardizzati che lo descrivono (*RepositoryObject*) e fornisce inoltre un set di servizi che consentono l'accesso e la condivisione dei contenuti e dei metadati tra entità organizzative in un ambiente federato.

Le specifiche OASIS *ebXML Registry Information Model* (RIM) definiscono i tipi di metadati ed i contenuti che possono essere memorizzati in un Registro ebXML, mentre le specifiche OASIS *ebXML Registry: Services and Protocols* (RS), definiscono i servizi erogati da un Registro ebXML ed i protocolli utilizzabili dai client del registro per interagire con tali servizi.

3.2 Registry Information Model (RIM)

L'*ebXML Registry Information Model* (RIM) definisce le classi e le loro relazioni utilizzate per rappresentare i metadati (*RegistryObject*).

In RIM la classe *RegistryObject* ed alcune altre classi sono derivate da una classe chiamata *Identifiable*, che permette di identificare gli oggetti mediante un attributo identificativo e consente l'estensibilità degli attributi prevedendo degli attributi dinamici denominati *Slot*.

Le sottoclassi di *RegistryObject* possono essere invece raggruppate nelle seguenti tipologie:

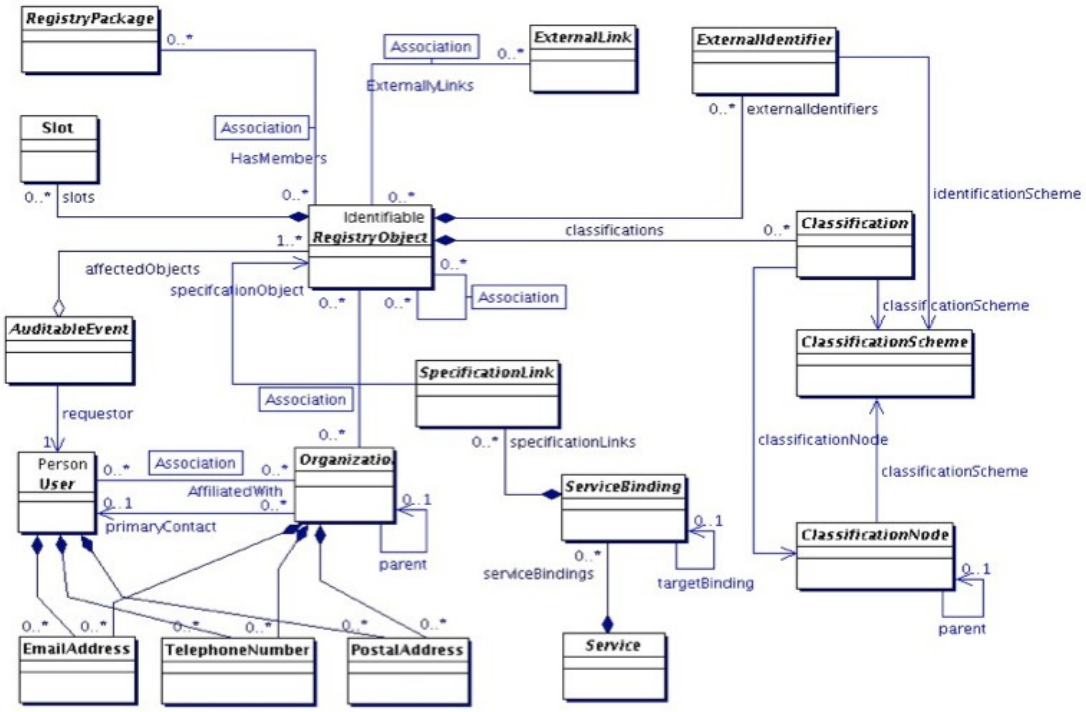
- *Core Information Model*: definisce le classi metadati nel modello di tipo core, incluse la classi base comuni;
- *Association Information Model*: definisce le classi che consentono alle istanze dei *RegistryObject* di essere associate le une alle altre;

- *Classification Information Model*: definisce le classi che consentono ai RegistryObject di essere classificati;
- *Provenance Information Model*: definisce la classi che consentono la descrizione della provenienza o la sorgente informativa di un *RegistryObject*;
- *Service Information Model*: definisce le classi che consentono la descrizione dei servizi;
- *Event Information Model*: definisce le classi che consentono la sottoscrizione agli eventi e le funzionalità di notifica come definite nell'*ebXML Registry: Services and Protocols* (RS).

3.2.1 Vista delle relazioni tra le classi

La figura seguente mostra una panoramica ad alto livello delle classe definite dal modello in termini delle loro relazioni di tipo "Has-A", come nei diagrammi di classe UML.

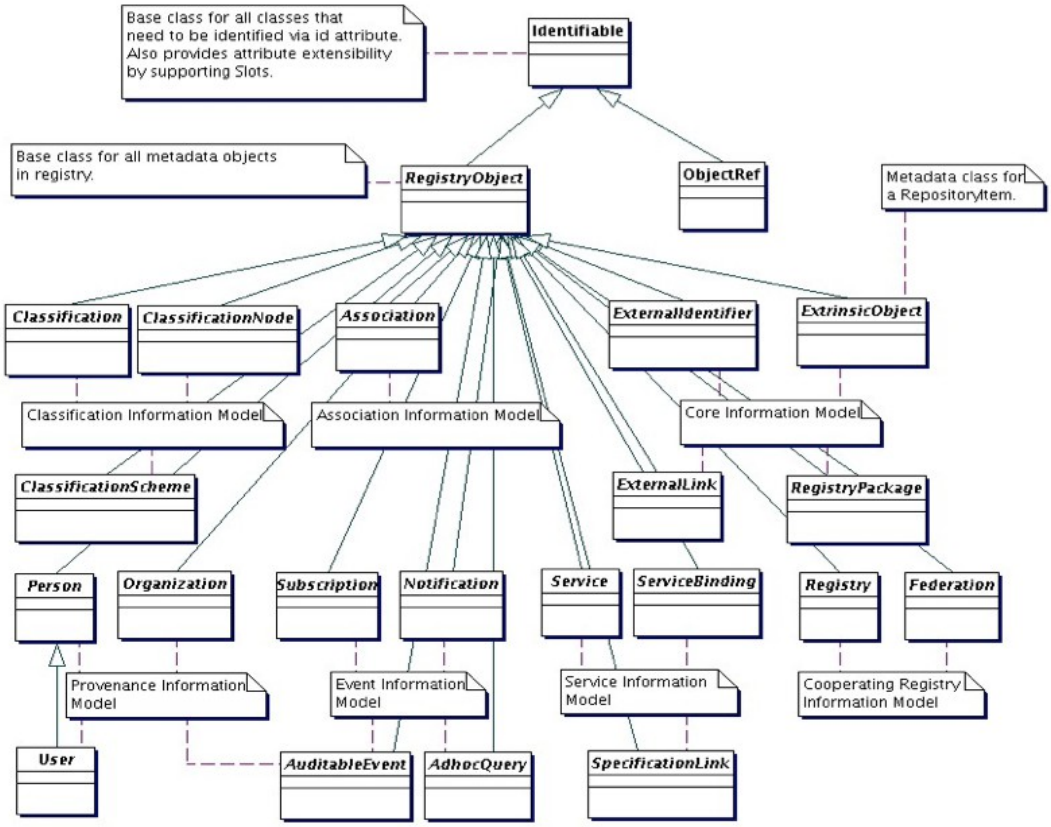
Non vengono mostrate le relazioni di tipo "His-A" (relazioni di ereditarietà) e nemmeno gli attributi delle classi.



3.2.2 Ereditarietà tra le classi

La figura seguente mostra le relazioni di ereditarietà tra le classi di tipo “Is-A”.

Non vengono mostrati altri tipi di relazioni, come la “Has-A” vista precedentemente, e nemmeno gli attributi delle classi.



3.3 Registry Services (RS)

Un registro ebXML è accessibile mediante le seguenti interfacce di servizio:

- Un’interfaccia di tipo *LifeCycleManager*, che fornisce un insieme di operazioni per la gestione del ciclo di vita dei metadati e dei contenuti all’interno del registro. Tra queste troviamo le operazioni di pubblicazione, aggiornamento, approvazione e cancellazione dei metadati e dei contenuti;

- Un’interfaccia di tipo *QueryManager*, che fornisce un insieme di operazioni per la ricerca e la lettura dei metadati e dei contenuti dal registro.

Le specifiche prevedono le interazioni con il registro avvengano esclusivamente mediante i protocolli SOAP e HTTP:

- con SOAP si consente ai client l'accesso al registro utilizzando il protocollo *SOAP 1.1 with Attachments*;
- con HTTP si consente ai client di tipo web browser l'accesso al registro utilizzando il protocollo *HTTP 1.1*.

3.3.1 Autenticazione e Autorizzazione

Un registry client dovrebbe essere autenticato dal registro per determinare l'identità associata e tipicamente questa sarà l'identità di un utente del Registro.

Una volta determinata l'identità, il Registro dovrà effettuare i controlli di autorizzazione e controllo degli accessi prima di consentire l'elaborazione delle richieste del client.

4 La tecnologia JAXR

4.1 Introduzione

Le JAXR (Java API for XML Registries) completano, insieme a JAXM (Java API for XML Messaging) e JAX-RPC (Java API for XML RPC), il supporto della piattaforma Java alle tecnologie dei Web Services. Come per molte altre tecnologie Java, che cercano di essere il più generico possibile, le JAXR consentono non solo l'accesso a registri UDDI ma anche ad altre tipologie di registri di servizi, come ad esempio ebXML.

Nel definire le specifiche JAXR sono stati tenuti in considerazione alcuni importanti obiettivi, tra cui:

- definire una API che fosse l'unione delle migliori caratteristiche delle principali specifiche per registri di servizi web;
- assicurare il supporto alle specifiche dominanti come UDDI ed ebXML;
- assicurare la sinergia con le altre tecnologie Java legate ad XML.

Sicuramente, l'obiettivo di assicurare il supporto UDDI é il primario scopo delle specifiche mentre il supporto ad ebXML appare più un tentativo di diffondere maggiormente quest'ultima tecnologia, inizialmente creata da SUN ed ora in mano al consorzio OASIS.

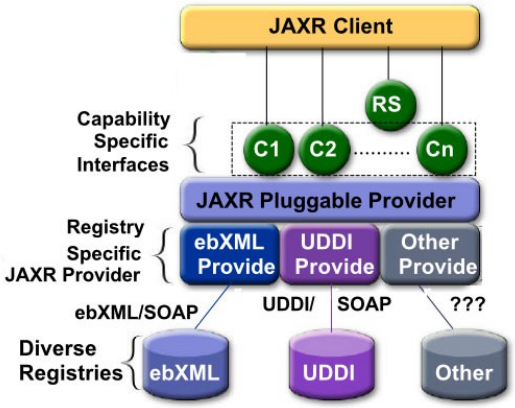
4.2 Architettura

Una implementazione JAXR è detta JAXR Provider e può essere di livello 0 o livello 1. Questo indica quali elementi sono supportati dall'implementazione. Ad esempio, il supporto ad UDDI è di livello 0. Questo significa che qualsiasi prodotto che dichiari la compatibilità a JAXR dovrà per forza supportare l'accesso ad UDDI. Il supporto ebXML è invece posto a livello 1, quindi i prodotti JAXR level 1 complaint supporteranno sia UDDI che ebXML.

Il supporto a diversi registri di Web Services avviene tramite l'utilizzo di un modello informativo comune ma estendibile che è stato mutuato dal modello informativo di ebXML con aggiunte prelevate da UDDI.

Questo ha consentito di risparmiare il tempo necessario a sviluppare un modello apposito e si traduce nelle specifiche come pochissime pagine dedicate alla mappatura di JAXR su ebXML.

Nella figura seguente è possibile vedere l'architettura generale di JAXR: un client utilizza sicuramente l'interfaccia RS (RegistryService). Questa è sempre presente e consente di ottenere informazioni sulla disponibilità delle funzionalità per lo specifico prodotto, come l'indicazione del livello di supporto dell'implementazione.



Le altre interfacce supportano le varie funzionalità, come la gestione del ciclo operativo e la gestione delle query.

Gli altri elementi dell'architettura sono il codice comune a tutti i registri, contenuto in "JAXR Pluggable Provider", e i componenti di accesso agli specifici registri ebXML, UDDI o altri.

L'implementazione delle API JAXR sono racchiuse in due package:

- *javax.xml.registry.infomodel* che contiene le interfacce per modellare le informazioni supportate da JAXR, come pure la loro interrelazione;
- *javax.xml.registry* contiene la definizione dell'interfaccia di accesso al registro.

4.3 Entità e classificazioni

Le specifiche JAXR definiscono una serie di entità che costituiscono il modello informativo su cui questo si basa. Come detto, queste sono entità presenti nel modello ebXML (anche i nomi sono sempre praticamente identici), con aggiunte mutuate da UDDI.

Le entità principali sono:

- *Organization*. Rappresenta una azienda, ad esempio quella che ha inviato informazioni sui propri servizi al registro. Può avere un riferimento alla "capogruppo" per modellare eventuali gruppi aziendali;
- *Service*. E' il servizio offerto dall'azienda;
- *ServiceBinding*. Sono le specifiche tecniche per l'accesso al servizio. Un servizio può avere più binding;
- *ClassificationScheme*. Rappresenta la tassonomia che può essere usata per classificare o categorizzare altri elementi del registro;
- *Classification*. Utilizzato per classificare concretamente un elemento del registro secondo uno specifico schema.

Attorno a queste entità ne esistono altre che consentono di collegarle tra di loro o di estendere il modello informativo.

Una delle caratteristiche più interessanti dei registri di Web Services, è infatti la possibilità di catalogare i servizi e le aziende secondo diverse tassonomie costruendo dunque un patrimonio informativo che consente di effettuare ricerche più puntuali.

Ad esempio è possibile restringere la ricerca ai soli venditori di libri online che operano sul territorio italiano ma che magari hanno sede in Svizzera.

JAXR supporta due diverse tipologie di tassonomia: interna ed esterna. Queste sono relative al registro: quando una tassonomia è riconosciuta da quest'ultimo, potrà essere

utilizzata come interna. In alternativa è possibile mischiare diverse tassonomie, o definirne di nuove, e creare così tassonomie esterne.

Solitamente le tassonomie vengono definite da organismi nazionali od internazionali, come nel caso di NAICS (North American Industry Classification System) - standard in uso nelle nazioni dell'America del nord.

Entrambe le tipologie di tassonomie hanno vantaggi e svantaggi, ad esempio le tassonomie esterne sono molto più indefinite e difficile da gestire e manutenere, mentre quelle interne sono predefinite e quindi forniscono un riferimento più solido.

Quasi tutti gli elementi del modello informativo di JAXR sono associabili tra di loro. Le associazioni non hanno una tipologia fissa ma si basano sull'astrazione di concetto, rappresentata nel modello informativo di JAXR come *Concept*.

Un *Concept*, in JAXR, può rappresentare un qualsiasi concetto e viene combinato con altri *Concept* per costruire un alberatura che rappresenti una tassonomia.

Azzardando un collegamento con DOM, si può dire che *Concept* di JAXR equivalga al *Node* di DOM. Con la differenza semantica che il primo rappresenta un concetto (p.e. l'Europa, l'Asia, la macchina o il pappagallo) mentre il secondo rappresenta un generico elemento di un documento XML.

4.4 Accesso ai registri

Come consuetudine nelle API Java, l'accesso alle informazioni avviene tramite una connessione restituita da una factory (altre API che utilizzano questo approccio sono JCA - Java Connector Architecture e JAXM - Java API for XML Messaging). La factory stessa può essere recuperata da un riferimento JNDI (Java Naming and Directory Interface, ed è il metodo consigliato) o attraverso la chiamata ConnectionFactory.newInstance().

In ogni modo, prima di ottenere una connessione al registro, è necessario impostare sulla factory ottenuta alcune proprietà indispensabili come l'URL del registro a cui si desidera accedere.

Una volta ottenuta la connessione è possibile:

- indicare se l'accesso al registro è sincrono od asincrono. Nel caso di accesso sincrono, tutte le chiamate alle API che prevedono l'interazione con il registro risulteranno bloccanti per il thread chiamante. In caso di accesso asincrono, la chiamata termina subito, eventualmente restituendo un oggetto JAXRResponse. Questa istanza non avrà ovviamente i dati richiesti in quanto verranno caricati con una sorta di Lazy-Loading (detto anche Futures Pattern), o all'atto dell'effettiva interrogazione di un metodo di GET, oppure tramite un thread apposito.

- impostare le credenziali dell'utente. Non è possibile accedere ai registri quali UDDI o ebXML senza dichiarare la propria identità: nei casi più semplici (UDDI) di utilizza la coppia utente/password, ottenuta tramite la registrazione presso il fornitore del servizio (IBM, Microsoft); in casi più complessi è possibile utilizzare certificati digitali X500 (in ebXML). Sicuramente comunque tutte le implementazioni JAXR possono fare uso di SSL e, nel caso di JAXR level 1, anche di criptare la comunicazione in quanto JAXR è allineato a JAAS (Java Authentication and Authorization Specification) e JSSE (Java Secure Sockets Extensions).

Ma cosa accade se in una applicazione è necessario accedere a diversi registri, ad esempio, sia ad UDDI che ad ebXML? JAXR consente di stabilire più connessioni contemporanee a diversi registri e di "federare" queste connessioni in una entità logica unica in modo, ad esempio, di poter eseguire funzionalità di ricerca su tutti i registri con una singola chiamata.

Sopratutto in questo caso, ma anche nel caso di singole connessioni, è importante, al termine delle operazioni con il registro, operare la chiusura della connessione. Questa alloca un numero significativo di risorse, quali connessioni di rete e thread che devono essere rilasciate al più presto per non compromettere le prestazioni.

4.5 Gestione del registro

Le JAXR consentono, ovviamente, la gestione delle informazioni presenti nel registro, come la creazione, l'aggiornamento e l'eliminazione di elementi. Inoltre è supportata l'interrogazione del registro per eseguire le ricerche di servizi ed aziende.

Ciascun elemento del registro è individuato univocamente da una UUID conforme al DCE 128 bit. Questa chiave è solitamente creata in modo trasparente dal registro anche se per alcuni registri (come ebXML) è possibile fornire dall'esterno la chiave da utilizzare.

Il ciclo di vita degli oggetti del registro inizia con l'operazione di creazione (l'interfaccia LifeCycleManager fornisce una serie di metodi di utilità per creare i differenti oggetti definiti nel modello informativo). Un oggetto creato non è ancora presente all'interno del registro e non può dunque essere oggetto di aggiornamenti o cancellazioni. Per inserire l'oggetto nel registro è necessario che questo passi attraverso un'operazione di salvataggio (save).

Una caratteristica interessante di JAXR è la possibilità di deprecare gli oggetti, con un concetto simile al tag @deprecated di Javadoc. Gli oggetti deprecati non consentono nuove referenze (come associazioni o classificazioni), ma continuano a funzionare normalmente.

Se da una parte l'interfaccia LifeCycleManager consente di gestire tutti gli elementi ad alto e basso livello del registro, può essere necessario concentrarsi più su elementi di business.

L'interfaccia BusinessLifeCycleManager contiene alcune importanti chiamate ad alto livello definendo una API simile alle Publisher API di UDDI. Con questa interfaccia non vengono introdotte nuove funzionalità ma viene dato un supporto agli sviluppatori UDDI che quindi si troveranno ad operare con API più familiari.

Un limite di JAXR in questa versione è che le operazioni sul ciclo di vita degli oggetti non sono applicabili a federazioni di connessioni. Non è, ad esempio possibile, creare la stessa entità direttamente in due registri differenti.

4.6 Interrogazione del registro

A differenza delle operazioni "dispositive" (come la creazione o la modifica di informazioni), le operazioni di ricerca avvengono in modo non privilegiato. Non è necessaria dunque l'autenticazione dell'utente.

L'interrogazione può avvenire in modo puntuale o tramite query. Nel primo caso viene utilizzata l'interfaccia BusinessQueryManager che, in modo similare a BusinessLifeCycleManager, ha una impostazione più ad alto livello e permette l'interrogazione delle interfacce più funzionali del modello informativo.

Molti metodi dell'interfaccia richiedono argomenti simili, alcuni tra i più importanti sono:

- *findQualifiers*. Definiscono le regole di confronto di stringhe, ordinamento e operatori logici sui parametri di ricerca;
- *namePatterns*. E' una Collection di stringhe che contengono nomi, anche parziali con eventuali wildcard come specificato nelle SQL-92 per la parola chiave LIKE;
- *classifications*. E' una Collection di Classification che identifica in quali classificazioni eseguire la ricerca.

Nel secondo caso è possibile utilizzare una query dichiarativa, tramite l'interfaccia DeclarativeQueryManager. Ad oggi l'unico standard supportato è una derivazione di SQL-92, in particolare dell'istruzione SELECT, esteso con le specifiche relative alle stored procedures.

Le query dichiarative sono supportate anche su connessioni federate ma sono una caratteristica opzionale dei provider JAXR 1.0.

5 JAebXR

5.1 Introduzione

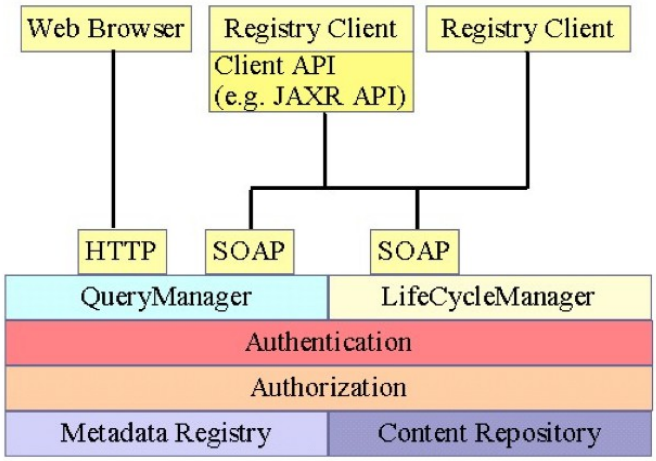
La libreria JAebXR (Java API for ebXML Registries) è stata sviluppata con l'intento di facilitare, standardizzare e ottimizzare le interazioni con registri di tipo ebXML.

La sua architettura deriva direttamente da quella di JAXR (Java API for XML Registries), incorporandola ed estendendone le funzionalità per garantire il supporto ai tipi di oggetti e servizi definiti dalle specifiche OASIS RegRep 3.0.

5.2 I client ebXML

Un client di un registro ebXML è un programma software che interagisce con il registro utilizzando i protocolli definiti dalle specifiche OASIS RegRep. Può essere di tipo GUI, un servizio software o un agent, e quindi essere eseguito su una macchina client o presentarsi come servizio di tipo web in esecuzione su di un server ed accessibile da un web browser.

Un client può accedere al registro direttamente o mediante una API, come la *Java API for XML Registries* (JAXR).

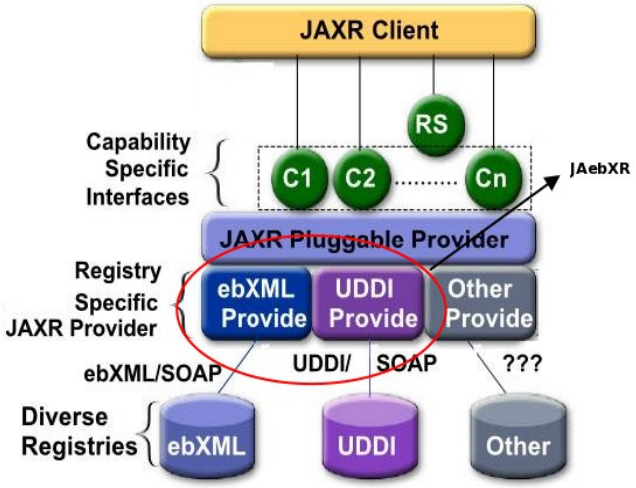


Qualunque sia la modalità scelta, l'accesso ai contenuti ed alle funzionalità offerte da un registro ebXML avviene tramite le due interfacce standard *QueryManager* e *LifeCycleManager*.

5.3 Architettura

Come già accennato, *JAebXR* si presenta come una API da utilizzare per la realizzazione di client ebXML e ripropone il modello architetturale di *JAXR*, estendendone le funzionalità al supporto degli oggetti OASIS ebXML RegRep.

Per tale ragione si può pensare a *JAebXR* come ad una implementazione di JAXR Provider di livello 1, poiché supporta anche i registri di tipo UDDI (livello 0).



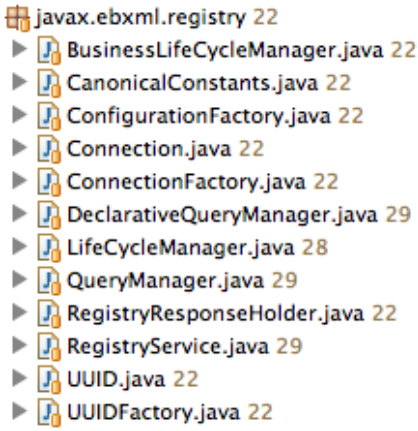
L'elemento che contraddistingue questa libreria è la sua completa indipendenza dalle particolari implementazioni dei registri ebXML.

Le interazioni con i registri ebXML avvengono, infatti, esclusivamente mediante invocazione in SOAP dei web service da questi esposti, servizi che devono essere presenti come da specifiche degli standard OASIS.

In tali circostanze, le interrogazioni e transazioni riguarderanno esclusivamente oggetti ebXML RIM, opportunamente incapsulati in elementi *JAXB* (Java Architecture for XML Binding).

5.4 Implementazione

L'implementazione è contenuta nel package *javax.ebxml.registry* che, come il package *javax.xml.registry* di JAXR, contiene la definizione delle interfacce di accesso ad un registro.



Come si può evincere dalla figura precedente, sono in particolare presenti le classi:

- *BusinessLifeCycleManager*
- *Connection*
- *ConnectionFactory*
- *DeclarativeQueryManager*
- *LifeCycleManager*
- *QueryManager*
- *RegistryService*

che non sono altro che implementazioni delle omonime interfacce di JAXR, estese ove necessario con la presenza di metodi specifici per oggetti ebXML RIM.

A completare il supporto per oggetti e registri ebXML RIM sono inoltre state implementate le classi:

- *CanonicalConstants*
- *ConfigurationFactory*
- *RegistryResponseHolder*
- *UUID*
- *UUIDFactory*

5.4.1 ConnectionFactory

La classe *ConnectionFactory* è la classe base fondamentale per la creazione di una connessione JAXR, che può avvenire mediante ricerca tramite JNDI (Java Naming and Directory Interface) o direttamente, grazie all'utilizzo del metodo statico *newInstance()*.

In questa sede è stato seguito il secondo approccio grazie al quale un client, tramite la proprietà di sistema *javax.xml.registry.ConnectionFactoryClass*, indica quale classe che implementa l'interfaccia *ConnectionFactory* (tipicamente quella fornita del provider del registro) debba essere istanziata mediante l'invocazione del metodo *newInstance()*.

La connessione vera e propria a un registro viene gestita completamente anche in termini di autenticazione, utilizzando le credenziali utente contenute in un apposito keystore.

Ciò significa che un'invocazione del metodo *createConnection()* ritorna un oggetto di tipo *Connection* già pronto all'uso.

5.4.2 LifeCycleManager

La classe *LifeCycleManager* è quella che, allo stato attuale dell'avanzamento sulla realizzazione della libreria, è stata maggiormente oggetto dell'attività di sviluppo.

Com'è noto, in JAXR tale classe è dedicata sostanzialmente alla gestione degli oggetti *Infomodel* di JAXR mediante la dichiarazione dei metodi astratti

- *createAssociation(),*
- *createClassification(),*
- *createClassificationScheme(),*
- *createConcept(),*
- *createEmailAddress(),*
- *createExternalIdentifier(),*
- *createExternalLink(),*
- *createExtrinsicObject(),*
- *createInternationalString(),*
- *createKey(),*
- *createLocalizedString(),*
- *createObject(),*
- *createOrganization(),*
- *createPersonName(),*
- *createPostalAddress(),*
- *createRegistryPackage(),*
- *createService(),*

- *createServiceBinding(),*
- *createSlot(),*
- *createSpecificationLink(),*
- *createTelephoneNumber(),*
- *createUser(),*

per quanto riguarda le operazioni di creazione, e mediante la dichiarazione dei metodi astratti

- *deleteObjects(),*
- *deprecateObjects(),*
- *saveObjects(),*
- *undeprecateObjects(),*

per le operazione di cancellazione, salvataggio e deprecazione.

In JAebXR, la classe *LifeCycleManager* implementa interamente l'interfaccia *javax.xml.registry.LifeCycleManager* e la estende al supporto per la gestione degli oggetti ebXML RIM mediante i metodi:

- *createAssociationType(),*
- *createClassificationType(),*
- *createClassificationSchemeType(),*
- *createConceptType(),*
- *createClassificationNodeType(),*
- *createEmailAddressType(),*
- *createExternalIdentifierType(),*

- *createExternalLinkType(),*
- *createExtrinsicObjectType(),*
- *createFederationType(),*
- *createInternationalStringType(),*
- *createLocalizedStringType(),*
- *createOrganizationType(),*
- *createPersonNameType(),*
- *createPostalAddressType(),*
- *createRegistryObjectListType(),*
- *createRegistryPackageType(),*
- *createServiceType(),*
- *createServiceBindingType(),*
- *createSlotType1(),*
- *createSpecificationLinkType(),*
- *createTelephoneNumberType(),*
- *createUserType().*

Cancellazione e salvataggio degli oggetti ebXML RIM sono invece implementati mediante i metodi

- *RegistryResponseType saveObjects(SubmitObjectsRequest req)*
- *RegistryResponseType deleteObjects(RemoveObjectsRequest req).*

Questi non sono altri che dei metodi wrapper verso il più generico metodo

RegistryResponseType submitObjects(RegistryRequestType req)

che implementa realmente le suddette operazioni mediante messaggi SOAP inviati direttamente al registro ebXML.

5.4.3 ConfigurationFactory

Gli aspetti di configurazione sono gestiti dalla classe *ConfigurationFactory*, che è stata realizzata seguendo il modello di design pattern creazionale denominato *singleton*.

Sono infatti presenti un unico costruttore privato, in modo da impedire l'istanziazione diretta della classe, e un metodo getter di tipo statico che restituisce una istanza della classe, creandola preventivamente o alla prima invocazione del metodo, memorizzandone il riferimento in un attributo privato anch'esso statico.

La classe si aspetta che le direttive di configurazione vengano indicate nel file di proprietà testuale denominato

ebxml.properties

e, una volta istanziata, ne effettua la lettura andando a valorizzare i vari attributi privati necessari per:

- effettuare una connessione di tipo JAXR ad un registro;
- istanziare un oggetto di tipo SOAPMessenger per l'invocazione autenticata del web service esposto dal registro ebXML.

5.5 Esempio di utilizzo

L'utilizzo della libreria è abbastanza semplice: è sufficiente innanzitutto predisporre adeguatamente il file di configurazione e poi, nel codice del client che si intende

sviluppare, invocare il metodo statico *newInstance()* della classe *javax.ebxml.registry.ConnectionFactory*.

Nell'esempio si propone una classe client minimale, che viene successivamente istanziata per eseguire delle semplici operazioni sul registro ebXML.

Inoltre si assume che:

- il registro ebXML utilizzato sia eRIC 3.2 e installato sulla stessa macchina;
- le credenziali di connessione al registro si riferiscano all'utente predefinito *RegistryOperator*, per il quale è disponibile il keystore che contiene il certificato

5.5.1 ebxml.properties

```
connectionFactoryClass=it.cnr.icar.eric.client.xml.registry.ConnectionFactoryImpl

aliasName=urn:freebxml:registry:predefinedusers:registryoperator
aliasPass=urn:freebxml:registry:predefinedusers:registryoperator

keystorePath=/opt/eric/3.2/data/security/keystore.jks
keystorePass=ebxmlrr
certificateType=JKS

registryURL=http://localhost:8080/eric/registry/soap
```

5.5.2 SoapRegistryClient.java

```
package it.cnr.icar.infse.common.registry;

import javax.ebxml.registry.BusinessLifeCycleManager;
import javax.ebxml.registry.Connection;
import javax.ebxml.registry.ConnectionFactory;
import javax.ebxml.registry.RegistryService;

import javax.xml.registry.BusinessQueryManager;
import javax.xml.registry.DeclarativeQueryManager;
import javax.xml.registry.JAXRException;

public class SoapRegistryClient {

   protected Connection connection = null;
   protected RegistryService service = null;
   protected BusinessLifeCycleManager lcm = null;
   protected BusinessQueryManager bqm = null;
   protected DeclarativeQueryManager dqm = null;

   public RegistryClient() throws JAXRException {
          openConnection();
   }
```

```
    public final void openConnection() throws JAXRException {
       if (service == null) {
          ConnectionFactory connFactory = ConnectionFactory.newInstance();
          connection = connFactory.createConnection();
          service = connection.getRegistryService();
          lcm = service.getBusinessLifeCycleManager();
          bqm = service.getBusinessQueryManager();
          dqm = service.getDeclarativeQueryManager();
            }
    }

    public BusinessLifeCycleManager getLifeCycleManager() {
            return lcm;
    }
}
```

5.5.3 SoapRegistryClientTest.java

```
package it.cnr.icar.infse.common.registry;

import javax.ebxml.registry.BusinessLifeCycleManager;
import javax.ebxml.registry.CanonicalConstants;

import javax.xml.bind.JAXBElement;
import javax.xml.registry.JAXRException;

import org.oasis.ebxml.registry.bindings.lcm.RemoveObjectsRequest;
import org.oasis.ebxml.registry.bindings.lcm.SubmitObjectsRequest;
import org.oasis.ebxml.registry.bindings.rim.FederationType;
import org.oasis.ebxml.registry.bindings.rim.RegistryType;
import org.oasis.ebxml.registry.bindings.rs.RegistryResponseType;

public class SoapRegistryClientTest {

    static RegistryClient registry = null;
    static BusinessLifeCycleManager lcm = null;

    public SoapRegistryClientTest() throws JAXRException {
       registry = new RegistryClient();
       lcm = registry.getLifeCycleManager();
    }

...
...

    public void testCreateDeleteFederation() throws JAXRException {
            String id = lcm.createUUID();

            FederationType f = lcm.createFederationType("testFederation");
            f.setId(id);

            JAXBElement<FederationType> eb = lcm.createFederation(f);

            SubmitObjectsRequest sreq = lcm.createSubmitObjectsRequest(eb);

            RegistryResponseType resp = lcm.saveObjects(sreq);
            System.out.println("Saved a new federation " + id);

            RemoveObjectsRequest rreq = lcm.createRemoveObjectsRequest(id);
            resp = lcm.deleteObjects(rreq);
            System.out.println("Deleted federation " + id);
    }

    public void testCreateDeleteRegistry() throws JAXRException {
            String id = lcm.createUUID();

            RegistryType r = lcm.createRegistryType("testRegistry");
            r.setId(id);
```

```
            JAXBElement<RegistryType> eb = lcm.createRegistry(r);

            SubmitObjectsRequest sreq = lcm.createSubmitObjectsRequest(eb);

            RegistryResponseType resp = lcm.saveObjects(sreq);
            System.out.println("Saved a new registry " + id);

            RemoveObjectsRequest rreq = lcm.createRemoveObjectsRequest(id);
            resp = lcm.deleteObjects(rreq);
            System.out.println("Deleted registry " + id);
    }

...
...

}
```

6 Bibliografia

Centro Europeo di Informazione su ebXML, "*OASIS / UBL - Universal Business Language Technical Committee*", http://www.ebxml.eu.org/It/UBL.htm

Java Community Process, "*JSR 93: Java API for XML Registries 1.0 (JAXR)*", https://jcp.org/en/jsr/detail?id=93, 2002

L. Marsili, Tesi di Laurea: "*Nuove proposte per l'interoperabilità aziendale: un'applicazione di eProcurement nel dominio dell'industria automobilistica*", LUISS Guido Carli, AA 2004/2005

OASIS ebXML Registry Reference Implementation Project, "*The freebXML Registry version 3.1 (OMAR)*", http://ebxmlrr.sourceforge.net/index.html

OASIS ebXML Registry Technical Committee, "*OASIS ebXML Registry 3.0 specifications*", http://docs.oasis-open.org/regrep/v3.0/regrep-3.0-os.zip

Presidenza del Consiglio dei Ministri / Consiglio Nazionale delle Ricerche, "*InFSE: Infrastruttura tecnologica del Fascicolo Sanitario Elettronico – Specifiche tecniche per l'interoperabilità dei sistemi territoriali di FSE*", http://ehealth.icar.cnr.it, Novembre 2013